PROCEEDINGS FROM THE 1990 WORKSHOP ON ISSUES OF INTEGRITY AND SECURITY IN AN ADA RUNTIME ENVIRONMENT

April 3-5, 1990

Orlando, Florida

IITRI

COMMITMENT TO EXCELLENCE

since 1936

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	Contract	**PR**	Project
G	Grant	**TA**	Task
PE	Program Element	**WU**	Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	See authorities.
NASA	See Handbook NHB 2200.2.
NTIS	Leave blank.

Block 12b. Distribution Code.

DOD	DOD - Leave blank.
DOE	DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	NASA - Leave blank.
NTIS	NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*

2. REPORT DATE
April 1990

3. REPORT TYPE AND DATES COVERED
Proceedings April 3-5, 1990

4. TITLE AND SUBTITLE
Proceedings from the 1990 Workhop on Issues of Integrity and Security in an Ada Runtime Environment

5. FUNDING NUMBERS

6. AUTHOR(S)
IIT Research Institute

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IITRI
4600 Forbes Blvd
Lanham, MD 20706

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
IITRI
4600 Forbes Blvd
Lanham, MD 20706

Ada Joint Program Office
Rm 3E114
The Pentagon
Washington, DC 20301-2080

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N/A

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Cleared for Public Release, Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document provides a summary of the results of the 1990 Workshop on Issues of Integrity and Security in an Ada Runtime Environment, which was held April 3-5, 1990 in Orlando, Florida. This section provides a background on the workshop and an introduction to each of the working groups. Appendix A is a compilation of the position papers that the conference attendees submitted. Appendix B contains the preliminary review of the current Catalog of Implementation Features and Options (CIFO) from a security perspective. The CIFO review, which was held as an evening meeting, was attended by a few of the workshop participants. Appendix C is a list of names and addresses of all the workshop participants.

14. SUBJECT TERMS
Security, Integrity, Ada RTE, Trusted Computer Base

15. NUMBER OF PAGES
88

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	

PROCEEDINGS FROM THE 1990 WORKSHOP ON ISSUES OF INTEGRITY AND SECURITY IN AN ADA RUNTIME ENVIRONMENT

April 3-5, 1990

Orlando, Florida

Sponsored by

IIT Research Institute
and
Ada Joint Program Office

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

The program committee would like to thank the following individuals for their contributions to the workshop by serving as working group chairs and for their contributions to these proceedings by preparing final reports for their working groups:

Dock Allen
John McHugh, Ph.D.
Charles McKay, Ph.D.
Richard Powers

The program committee would also like to thank all of the participants for their contributions.

1.0 INTRODUCTION

This document provides a summary of the results of the 1990 Workshop on Issues of Integrity and Security in an Ada Runtime Environment, which was held April 3-5, 1990 in Orlando, Florida. This section provides a background on the workshop and an introduction to each of the working groups. Appendix A is a compilation of the position papers that the conference attendees submitted. Appendix B contains the preliminary review of the current Catalog of Implementation Features and Options (CIFO) from a security perspective. The CIFO review, which was held as an evening meeting, was attended by a few of the workshop participants. Appendix C is a list of names and addresses of all the workshop participants.

1.1 Background

The objectives of ~~the 1990~~ Workshop on Issues of Integrity and Security in an Ada Runtime Environment were:

1) to identify and discuss the security and integrity issues related to the Ada runtime environment; [illegible]

2) to bring together and create some synergy among the security/integrity and Ada specialists in order to adequately address security and integrity issues related to the Ada runtime environment.

The workshop was jointly sponsored by IIT Research Institute and the Ada Joint Program Office.

1.2 Working Group Organization

At the onset of the workshop participants were asked to participate in one of the following working groups:

1. Ada Runtime Environment

2. Applications of Formal Methods to Security and Integrity of a Trusted Ada Runtime Environment

3. Issues of Access Control in a Distributed Environment With Persistent Data.

The Ada Runtime Working Group, led by Ms. Dock Allen of Control Data Corporation and Mr. Richard Powers of Texas Instruments Defense Systems and Electronics Group, addressed the security and integrity issues directly related to the Ada runtime environment and reviewed the most recent CIFO entries with respect to security and integrity concerns.

The Applications of Formal Methods to Security and Integrity of a Trusted Ada Runtime Environment working group, led by Dr. John McHugh of Computational Logic, Inc., discussed the applications of formal methods to security and integrity of trusted Ada runtime environments.

The Issues of Access Control in a Distributed Environment With Persistent Data working group, led by Dr. Charles McKay of the University of Houston at Clear Lake, addressed the issues of access control in a distributed environment with persistent data.

The three working groups were each composed of approximately eight representatives from Government, industry, and academia.

2.0 ADA RUNTIME WORKING GROUP

The Ada Runtime Working Group focused on the issues of security and integrity that are a result of Ada runtime environments, as opposed to the general issues of security and integrity that apply to all languages. The focus was primarily on the runtime itself, rather than the entire Ada language or pre-runtime tools.

The working group contained 10 participants, representing both the government and industry. The participants were:

Dock Allen (Control Data),
Edward Beaver (Westinghouse ESG),
George Buchanan (IIT Research Institute),
Michael Diaz (Motorola GEG),
Clarence "Jay" Ferguson (National Security Agency),
Ed Gallagher (CECOM Center for Software Engineering),
Mark Kraieski (McAIR/LHX),
Nina Lewis (Unisys),
Fred Maymir-Ducharme (IIT Research Institute),
John Perkins (PRC),
Richard Powers (Texas Instruments),
Ken Rowe (NSA/NCSC).

The working group covered the following topics:

Issues and Assumptions
Definition of a Process for the Workshop
Identification of General Threat Types
Definition of a Working Model and its Interfaces
Analysis of the Security Issues for Typical Ada Runtime Features
Allocation of Security Requirements to a Typical Ada Runtime
Ada Features Required to Build Integrity Into Applications Recommendations

2.1 Issues Definition

At the beginning of the workshop, the group identified a set of issues that were significant with respect to integrity/security/Ada runtimes. Although there was not sufficient time to address most of the issues, we present them as background. Having identified the issues that the participants were concerned with, we focused the rest of the workshop on the Ada Runtime issues.

How do security and integrity issues differ across the domain of systems?

The domain ranges from single/stand-alone type processor system to large, distributed systems of main-frame class machines.

How are security and integrity requirements to be applied across this range?

Compiler trustedness:

How can the trustedness of a compiler be established, so that the code generated by the compiler can be trusted?

Controlling development of the Ada Runtime:

One major question was how "trusted" the Ada runtime had to be. This is related, in part, to the security boundary issues, and whether or not the Ada runtime is shared between subjects. Another major concern is using a commercial Ada runtime. If the Ada runtime was developed outside of a particular program's control, how can its trustedness be established?

If there are packages or services that should be restricted, how does the pre-runtime (compilation system and support tools) enforce these restrictions?

Many of the packages needed to build embedded systems can interfere with system integrity if used unwisely; yet these are needed to deploy some real-time systems. Compilation systems that support limited access to such packages are clearly needed.

Protecting classified data from disclosure:

If data has been passed on the stack, or has been placed on the heap, the data image can persist after the memory is released. Some form of scrubbing of released stack and heap space may be required.

Protection of classified algorithms:

Protection from disclosure should include code, as algorithms can be classified.

Control over the placement of classified information:

To support secure operation, the placement of algorithms and data in memory must be controlled. For code, and data defined at compile time, this is an issue for the Compilation system and its tools. However, the runtime controls placement of dynamically allocated data. This may conflict with security needs.

Elimination of unused code:

Code that is never executed must not be loaded for certain classes of systems. This has implications for the runtime, as well as for the compilation; the runtime features which are not used should be configured out.

Authentication of code:

Code should be authenticated when it is loaded; this is not typically an Ada runtime responsibility. The code supplied by the Ada vendor may require authentication, including the Ada runtime itself.

Security Boundaries:

A major issue discussed in the working group was where the boundaries between subjects is in a secure system. Part of the group recommended that the boundary be an Ada program, so that a secure multi-programming environment could be built to enforce access controls.

Another part of the group felt that security must exist within an Ada program, and that there were some applications, such as communication servers, where the expressiveness of Ada tasking is required to produce a good application. In this example, the security boundary should be at a lower level than the Ada program, perhaps between tasks.

Yet a third party indicated that a hybrid approach was needed. Clearly, there was no consensus on this issue. There was some agreement that the multi-programming approach would work; there was not consensus that it would be sufficient.

If security boundaries are lower than the program level, what are they?

How does an Ada task relate to a process or subject?

What is the trusted computing base (TCB) with respect to the Ada Runtime?

How much of the TCB, if any, is in the Ada Runtime?

Does the Ada Runtime run "on top of" a TCB, or does some portion of the TCB include some of the Ada Runtime functions. This is in part related to the question of the security boundaries. In addition, the typical size of Ada Runtimes would seem to preclude having the entire ARTE be a TCB.

What threats should the Ada Runtime be responsible for handling?

This relates, in part, to the issue of where the security boundaries are. It also includes questions of how the Ada runtime can protect the system from malicious and intentional compromise or corruption, and what vulnerabilities may arise from the runtime itself.

When the environment includes an Ada runtime and an operating system, what additional issues arise?

An Ada Runtime implemented "on top of" an operating system generally allocates some of its functionality to the OS.

How does a TCB fit into this relationship?

Does Ada tasking present security/integrity issues?

The size of the Runtime to support Ada tasking is a concern, with respect to code validation. Preservation of the * (star) property among tasks of different levels may present problems.

What denial of service issues are of concern in Ada?

Most of the issues identified involved some form of resource gluttony or abuse:

A task/program can consume the heap, which will prevent other tasks/programs from executing;

A task/program can consume the CPU;

A task/program can cause fragmentation of the heap;

A task/program can force excessive garbage collection by fragmenting the heap;

A task/program can consume channel/bus capacity;

An Ada runtime is allowed to keep a pointer for each terminated task (instead of one common pointer), which can consume the heap for long-running programs.

What should the runtime/TCB do in response to a security violation?

In an embedded system, terminating the program may not be an acceptable response due to mission requirements, whereas this might be appropriate in a mainframe environment.

How can Ada programs access low level hardware features without introducing security/integrity problems?

Ada permits programs to access interrupts, when supported by the runtime, and to access low level hardware interfaces. This allows applications to do functions which are unique to a particular system, such as providing device drivers for special-purpose devices.

Can the runtime mediate this access such that the application can provide these low level service without compromising system security and integrity?

Is the Ada Runtime involved in encryption?

If encryption is an application, from the runtime perspective, how can that application be trusted if the Ada runtime is not trusted?

How does reconfiguration of processor resources impact system integrity and security?

If an Ada program spans processing resources, what impact does reconfiguration have on the integrity and security of the system?

The group noted that the Ada LRM says nothing about a partially functional Ada program.

How do tailorable Ada runtimes affect system integrity and security?

The need to keep Ada runtimes small comes from several pressures. In addition to saving memory, which is still a major concern in some systems, the smaller Ada runtime is more amenable to verification.

How can the integrity and security of an Ada runtime be insured, in the face of arbitrary subsets of functionality?

How can security requirements be verified in complex systems?

> Given that it is not possible to perform formal proof on a reasonable Ada runtime environment, what other techniques are available to validate that system security requirements have been met?

What features are required in an Ada runtime such that a distributed security policy can be implemented on top of it?

> Distributed system services are often built upon, or integrated into, an Ada runtime. If these services have security requirements, what features does the Ada runtime need to support?

This list just scratches the surface of the security and integrity issues. The working group was not able to do more than list most of the issues. The remainder of the workshop was spent on analysis of the runtime, and its impact on integrity and security.

2.2 Assumptions

1. The Ada compilation system(s) can limit programmer access to software packages and interfaces. This assumption allowed the group to be less concerned with interfaces that may be necessary to portions of the application builders, but that introduce security/integrity risks.

2. Trusted interfaces will validate requests.

3. The underlying hardware has sufficient support to build a trusted system. This assumption allowed the group to sidestep the issue of what hardware support might be required.

4. The group was not concerned with protecting erroneous "subjects" from damaging themselves.

5. If multiple programs "with" a package, they get separate copies. This assumption allowed certain assumptions to be made about a multi-programming approach to supporting security and integrity.

6. The group was not concerned with programs which span processors.

2.3 Process

The working group used a process which involved adopting a working model for the system, a typical Ada runtime architecture, and evaluating the security and integrity issues within this frame work. Since it would not be possible to do a thorough analysis of the domain, the group used the strategy of using representative sets of issues applied to working models. The steps followed were:

1. Identify a set of representative system threats;
2. Adopt a working model for the architecture, Ada Runtime, and TCB;
3. Adopt a working model for an Ada Runtime;
4. Identify threats that originate in, or are exacerbated by, the ARTE;
5. Identify threats that are handled by the ARTE;
6. Determine the requirements that security and integrity would levy on the ARTE;
7. Prepare a set of recommendations for future research, analysis, and prototyping.

2.4 Identification of Threats

This activity identified a subset of system threats, in order to provide a focus for the remaining analysis. The threat countermeasures identified were:

Protection of data and code from:

1. Disclosure of classified information;
2. Corruption;
3. Loss;
4. Intentional or inadvertent denial of service;
5. Sabotage or errors during development of the applications of the system software, including the ARTE;
6. Presence of unauthorized code or data;
7. Protection of the execution environment, including the hardware state and the Ada runtime state;
8. System level issues, such as encryption.

2.5 Working Model and Interfaces

In order to explore the issues of Ada and security, the working group adopted a working model for the TCB. This is not the only possible model, but serves as a framework for discussion. In this model, it is assumed that the functions of the TCB are provided by a combination of hardware, Ada Runtime Environment (ARTE), the extended runtime library (XRTL), and the application.

We assumed that the hardware provided sufficient support for secure a operation; this usually means that AT LEAST a user/kernel separation is maintained, or that a capability based architecture is used.

The ARTE provides the runtime services needed by Ada, such as tasking and memory management. The model assumes that much of the TCB is *implemented within*, or underneath the ARTE. The model does not assume any particular relationship between the ARTE and the TCB, except that they appear as a unit to the XRTL and application software. If the system is built upon an existing operating system, this would also be considered part of the ARTE, from the perspective of the model.

The XRTL includes those functions typically thought of as system services, such as I/O drivers. In some systems, support for distributed processing is an XRTL feature. Many runtime extensions for real-time software, such as specialized memory managers, can also be XRTL services.

The model allows for the case where some of the TCB functions are provided by XRTL services, and even by the application itself. The model is illustrated in Figure 2.5-1.

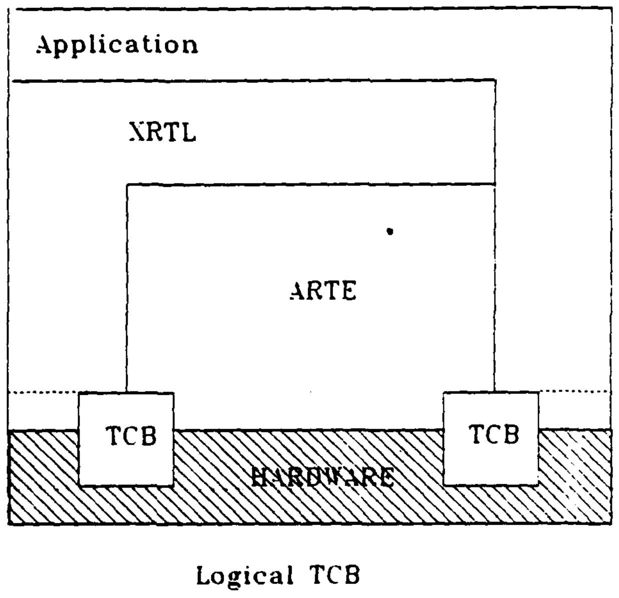


Figure 2-1 Working Model Software Architecture.

In order to evaluate the Ada Runtime using this model, we developed a list of the interfaces in the model, along with a definition of what each interface consisted of. The interfaces are summarized in TABLE 2-1.

TABLE 2-1
INTERFACE SUMMARY

Application to XRTL	The XRTL consists of Ada packages which provide system-level services for the application; the ARTEWG CIFO defines several XRTL services. The XRTL interface is defined by the packages specifications for those services. The interface is explicit, consisting of Ada statements which invoke the services.
Application to ARTE	The ARTE interface is implicit; the vendor defines the interface, and the compiler generates calls to the runtime in response to Ada statements in the source. The LRM, plus any vendor supplied pragmas represent the interface to the ARTE.
Application to Hardware	This interface is the Instruction Set Architecture (ISA) of the system. The application access is via code emitted by the compiler, and is often limited to those instructions and addresses that are available in "user mode", in systems which have a user/kernel separation in hardware.
XRTL to ARTE	This is the same as the Application to ARTE interface. Some additional packages and pragmas may be provided to the XRTL, on a limited basis, to provide access to lower level runtime interfaces.
XRTL to Hardware	This is the same as the Application to Hardware interface, except that portions of the XRTL may run in "kernel mode", and have access to the complete ISA.
ARTE to Hardware	This is the same as the Application to hardware interface, except that much of the ARTE will execute in "kernel mode".

2.6 Example of an ARTE

The working group developed a list of functions typically supplied by an ARTE. This list was then used to analyze parts of the ARTE (and functions of Ada) that might be considered a risk to security and integrity.

Because the group had not defined where the security bounds should be drawn (i.e.

what is a subject?), the first pass through the list assumed that tasks within a single program might be multiple subjects. This led to the numbered list that follows. The second pass through the list assumed that multi-programming was used, and that each Ada program was a subject. This removed several issues. The remaining issues are indicated with an asterisk ("*") in the list. Finally, a pass was made through the list to decide which of the remaining issues might be removed with an implementation of multi-programming that better comprehends the security/indicated issues. The items that remained after this process are indicated by a hash mark in front of the asterisk ("#*").

2.7 Analysis of ARTE Functions

Typical functions of the ARTE and possible security/integrity problems/issues:

ARTE 1 - Ada Exception Management

Functions: Raising exceptions and propagating them to applications.
Problems and Issues:

1. Raising an exception in a rendezvous can affect the execution of another task.

2. Anonymous and misleading exceptions can cause several problems.

ARTE 2 - Storage Management

Functions: Support for the allocation ("new") and deallocation (UNCHECKED_DEALLOCATION) of user data. Responsible for some portions of stack management.

Problems and Issues:

*3. Heap Creep (fragmentation due to allocation/deallocation leading to denial of service).
*4. Gluttony (allocating all available memory).
*5. Object reuse (i.e. support needed for scrubbing).
#*6. Loss of CPU time due to excessive garbage collection (due to fragmentation).
*7. Stack Scrubbing.
#*8. Register Scrubbing.
#*9. Address Space Management.

ARTE 3 - Task Management

Functions: Task creation, activation, termination, completion, abortion. Support for various forms of rendezvous.

#*10. CPU Gluttony.
11. Abortion (of a visible task).
12. Abortion may not be immediate/timely.
13. Rendezvous. Many opportunities for problems exist when two tasks communicate.
#*14. Task Gluttony (i.e. creating tasks until the system can no longer support and new tasks).
#*15. Refusal to terminate/complete.
16. Priority inversion.
17. Priority inheritance during a rendezvous.

ARTE 4 - Time Management

Functions: Support for the delay statement. Support for package CALENDAR.

#*18. Delay implies a covert timing channel.
#*19. Numerous short delays can tie up the ARTE handling delay completion, and lead to a denial of service.

ARTE 5 - Input/Output

Functions: Support for predefined I/O packages.

#*20. A file system implies many security/integrity vulnerabilities.
*21. Buffer reuse/scrubbing.

#*22. LOW_LEVEL_IO implies access to underlying hardware devices.
*23. Buffer flushing.
#*24. Channel/device gluttony.
*25. Buffer gluttony.
26. Devices/files/buffers could be used as covert channels.

ARTE 6 - Initialization

Functions: Responsible for elaboration, elaboration checks, and system initialization.

#*27. Hardware initialization.
#*28. Memory scrubbing.
29. Elaboration failures can affect all tasks in a program.

ARTE 7 - Shutdown

Functions: Concerned with program completion.

#*30. Scrubbing program resources upon exit.
#*31. The inability for a main procedure to abort itself (and all of its tasks.)

ARTE 8 - Compiler Support

Functions: Compiler dependent routines.

#*32. Must be evaluated on a compiler by compiler basis.

ARTE 9 - Interrupt Management

Functions: Responsible for handling machine interrupts/exceptions.

#*33. Interrupts being disabled/masked for too long can leave to denial of service.
#*34. Interrupts can be used as covert channels.

2.8 Allocation of Security Requirements to a Typical Ada Runtime

Based on the analysis conducted, the working group allocated the following security requirements to the ARTE:

Detect, prevent, recover from, and report various forms of gluttony, including CPU and memory.

Manage the address space protection features of the hardware;

Implement scrubbing/object reuse for the heap, stack, registers, external storage (where the ARTE provides the I/O services for that storage), and program memory;

Provide trusted device services, including file management;

Monitor covert channels which are involve the runtime, such as covert timing channels in the Ada delay;

Provide a means for an Ada main program to terminate itself or be aborted;

Detect, report, and recover from deadlock and starvation;

Support a trusted audit log; this could be implemented by the ARTE; alternately, it is implemented outside of the ARTE and used by the ARTE. This was felt to be an implementation decision.

These do not represent a complete set, by any means, but serve as an example of what would be required.

2.9 List of ARTE Features to Support Integrity

The group next discussed a list of ARTE features that could assist in building high-integrity applications. The ARTE could export interfaces to all programs to:

1. Define a memory partition within a program.
2. Clear memory (heap, stack, and registers) at certain times. This would allow the program to control when scrubbing occurred.
3. Initialize memory.
4. "Freeze" memory (i.e. make it read-only) at some point during execution. This would allow the program to calculate values, then insure then they were not accidentally changed.
5. Report stack history (trace back) at runtime.
6. Log exceptions.
7. Detect a read of an undefined object.
8. Terminate/abort the main program. Currently, the main may unable to terminate because other tasks are still running.
9. Limit the view of the standard I/O packages. Currently, a user gets all of the I/O package, such as text_io, when the package is with'd. For integrity reasons, it would be desirable to limit the capabilities which a particular user obtained.

Several issues were also discussed that were determined to be more of a toolset issue than an ARTE issue. The toolset could aid the application builder by:

1. Allowing total control over elaboration order.
2. Allowing total control over parameter passing mechanism. This eliminates an area of unpredictability in the program behavior when non-scaler objects are used as parameters.
3. Allowing control over order of evaluation of expressions.
4. Limiting the use of language features for parts of a program.

2.10 Recommendations

1. Evaluate the feasibility of using host tools to check programs for secure and high integrity use of Ada.
2. Evaluate the ARTEWG CIFO from a security and integrity perspective.
3. Propose and evaluate alternate TCB software architectures.
4. Propose and evaluate alternate approaches to subject boundaries (e.g. programs versus lower than programs, functional versus lexical).
5. Evaluate where current compilers do not efficiently support Ada features which are valuable for security and integrity.
6. Identify hardware support needed for or beneficial to proposed secure software architectures.
7. Develop guidelines for use of Ada in secure/high integrity systems.
8. Examine and recommend approaches for tools to control use of Ada/XRTL features.
9. Continue to evaluate/identify/elaborate security-related ARTE and Ada issues and solve the problems.
10. The group strongly supports any Ada 9X effort to provide more predictability and formalism for Ada in the interest of security.
11. Foster research addressing formal verification of concurrent Ada.
12. Develop guidelines for CIFO use on secure/high interest systems.

3.0 APPLICATION OF FORMAL METHODS TO SECURITY AND INTEGRITY OF A TRUSTED ADA RUNTIME ENVIRONMENT

Being formalists, even if only temporarily, or by osmosis, we feel a need to question some basic premises. It is unclear to us that there are specifically identifiable Security and Integrity issues associated with run time environments for Ada, per se. We see a variety of issues associated with Ada code that becomes part of the TCB, whether this code represents a trusted application, a RTS, or an operating system kernel. We feel that the development of lengthy lists of very specific issues indicates a failure to comprehend the overall problem. Accordingly, the Formal Methods group has concentrated on process.

We believe that our results establish a framework that can be used to place the detailed issue lists of the other working groups in an appropriate context. Before presenting framework, we will discuss a few general issues. The report continues with an outline of the research issues that provide the bulk of the section. The research issues constitute framework mentioned above. These are followed by a couple of issues that are seen as technology transfer rather than research. The section concludes with a roadmap that outlines a research agenda and a position statement on the way in which the agenda should be executed.

General Issues:

First, we would like to acknowledge the participation of the members of the working group (listed below). Without their efforts, this work could not have been developed. As group leader, I (John McHugh) accept responsibility for any sins of omission or commission.

John McHugh, (Computational Logic, Inc.)	*Group leader
James Alstad, (Hughes Aircraft Company)	
Paul Cohen, (Martin Marietta)	
Steven Goldstein, (IIT Research Institute)	
Ann Marmor-Squrires, (TRW)	
John Perking, (DRC)	
John Shultis, (Incremental Systems Corporation)	

We are less than comfortable with the conceptual basis of the workshop. Our collective experience leads some of us to feel that, while we understand the TCSEC reasonably well, we are not completely comfortable with it as a basis for a formalization of security in a real sense. With an increasing tendency towards the formulation of mission-specific security policies and the notion of trusted applications, we feel that a more flexible and general framework is appropriate as a formal basis. Because much of the discussion of the workshop comes into the category of trusted applications, we feel that it is appropriate to raise this question here. We are even less comfortable with the definition of integrity. We adopted both a majority and minority definition.

> Integrity is the assurance that the response of a system to a stimulus is in conformity with the system specification.

This is so comprehensive as to be equivalent to functional correctness. The minority definition is an attempt to restrict the scope to computer and software issues.

> Integrity is the agreement of system outputs with the specification.

A third definition was suggested by Steve Goldstein. This couches integrity in terms of data corruption with the notion that unauthorized modification reduces the integrity of the data. All of these are distinct from hierarchical notions of Biba and others.
To me, none of these are particularly satisfying. Integrity is not a binary quantity, but we seem to lack a metric for quantifying integrity. There is no basis for the establishing the integrity of externally developed data or for specifying the effects of software on the relationship between the integrity of input and output data. There is no basis for establishing the integrity metrics for software.

In the discussion that follows, the following definition is used to define formalism.

> Formalism is an unambiguous expression of the paradigm and vocabulary that define a semantic model.

The following provides a list of research issues and a basis for categorizing issues:

The following are research issues.

I. What methodologies are suitable for using formal methods in the development and maintenance of trusted Ada runtime systems?

   A. What are the concepts that need to be axiomatized?
      Partial answer - rts sensitized
      multi-processing
      real-time
      Inter-program communication
      (this list needs S & I concepts added)

B. What is a good formal language for expressing security and integrity properties?

Position: The language must cover, at least, execution environment and Ada dynamic semantics.

1. Is a logic more expressive than first order logic desirable?
2. How can inherent and/or deliberate ambiguities and consistencies in Ada be expressed in the formal language?

C. What are the appropriate paradigm and vocabulary? Ontology?

1. Do we need an execution model as well as an Ada level model?
   Position: Yes
   a. What is an appropriate vocabulary for specifying an execution environment?
   b. What is an appropriate vocabulary for specifying an Ada dynamic semantics?
      Position: If these vocabularies are different, there will be consequences to investigate. The investigation will need to consider both domains.

D. What are appropriate Formal Methods for Security and Integrity in Ada.

E. What is a formal language that flows down well into system/software implementation languages such as Ada.

F. What tools are required to support the above methods and methodologies.

II. Are there RTS-specific issues?
Position: We feel that TCB specific issues exist that affect the RTS or even applications if they are part of the TCB. There are no RTS issues, per se.

III. What is the relationship between Application Security and Integrity and the RTS?
Position: This question is addressed with the "Onion Skin" diagram with appropriate overlays.

IV. Is there an incremental approach to the development of formalisms, methods, and tools?
Position: Yes

A.What useful short term research results can be obtained through incomplete and/or approximate formalisms?

Position: E.g, How we deal with ambiguous and incomplete run time models?

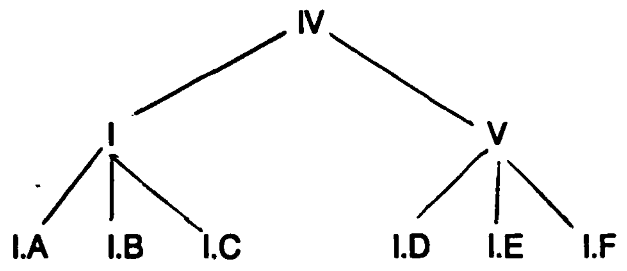
The following are technology transfer issues.

V. How should Formal Methods be introduced into practice.

VI. What we can say today about dealing with the informality of existing languages, systems, and specifications?

Position: Use safe subsets. Work has been done on this by TRW (ASOS), ORA (Penelope), CLI (AVA), NPL (Low Ada).

A ROADMAP FOR RESEARCH



POSITION

Incorporation of formal methods in software engineering practice requires a cooperative effort involving practitioners in the design and engineering of formal methodology and greater understanding and appreciation of software practice on the part of formal methods researchers.

We, therefore recommend a multi-threaded approach involving teams of researchers and practitioners, preferably situated in the application development environment, to negotiate approximate solutions of real utility and strategies for extending them to progressively more complete solutions.

4.0 ISSUES OF ACCESS CONTROL IN A DISTRIBUTED ENVIRONMENT WITH PERSISTENT DATA

The following individuals were members of the Issues of Access Control in a Distributed Environment With Persistent Data working group:

Charles McKay * chair
William R. Worger (US Army)
Sue Le Grand (Planning Research Corporation)
Jeffrey L. Grover (Georgia Tech Research Institute)
Capt. Robert Pierce (US Air Force)
Ann Maymor-Squires (TRW)

The final report of this working group was divided into three parts: the context of this workshop and working group, the major issues addressed by this working group, and their recommendations.

The context of this workshop and working group is introduced in Figure 4-1. As shown, the external oversights and other stimuli that synergistically affect national issues of policy, standards, and research and development will impact the system security policy for any project. (A project is indicated by the rectangle in the middle of the figure.) Just as the perspectives of both users and acquisition personnel are influenced throughout the project life cycle by the system security policy adopted for the project, these perspectives also reflect the influences of the vendors and supply communities (shown at the bottom of the figure) that provide a portion of the stimuli (shown at the top of the figure). Specifically, the compilers, runtime environments, and other tools and components provided by these vendors and supply communities will directly and indirectly influence the expected capabilities and delivered items throughout the project's life cycle.

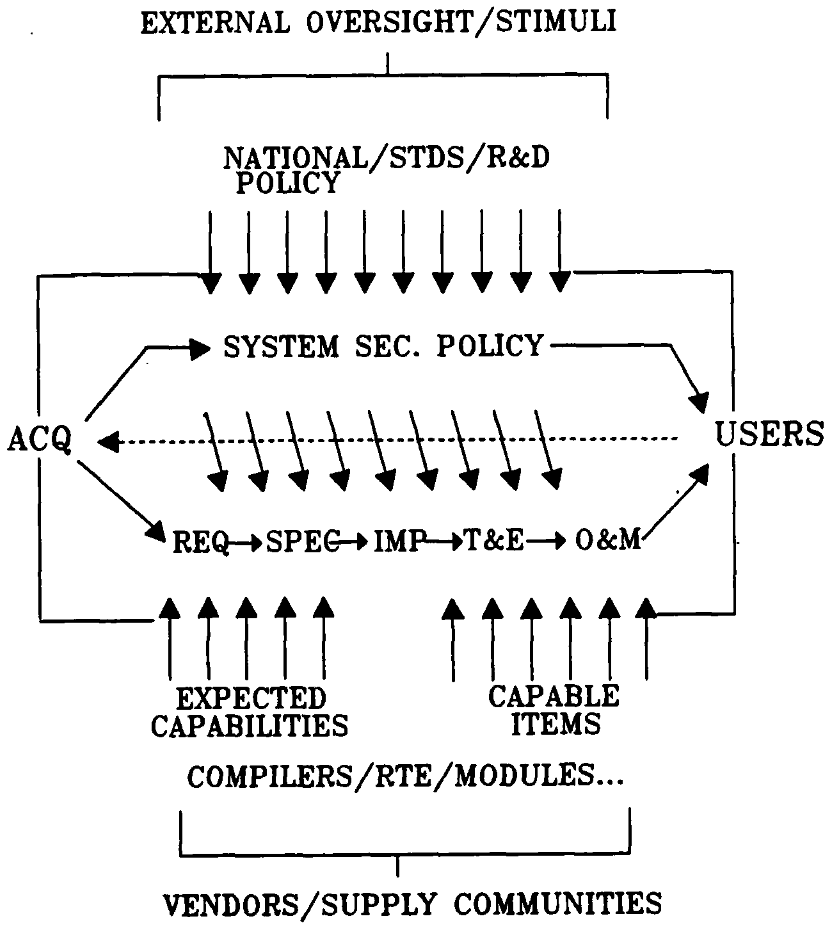


Figure 4-1. Overview of Working Group's Position.

The context of this workshop reflects issues at the top of Figure 4-2. In particular, the workshop focused on the research and development issues that might facilitate real progress in future projects of national importance. This working group focused on the capabilities needed from the vendor and supply community that are unlikely to be available in a timely fashion unless these research and development issues are properly addressed.

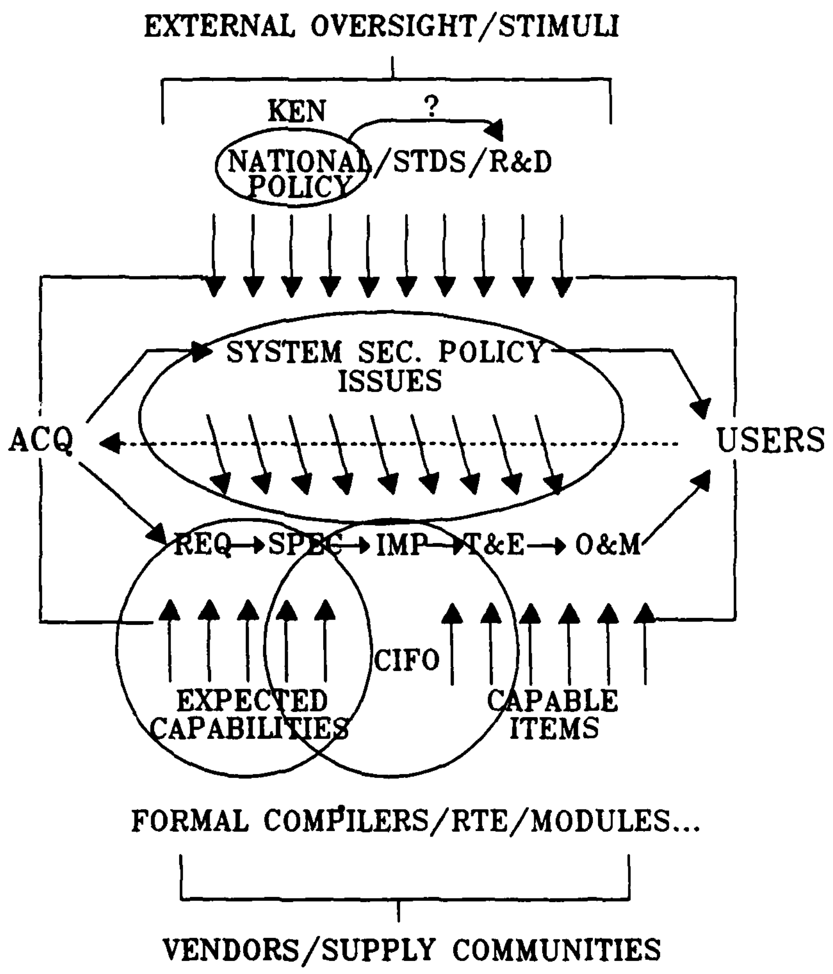

EXTERNAL OVERSIGHT/STIMULI
KEN
?
NATIONAL POLICY
/STDS/R&D
SYSTEM SEC. POLICY ISSUES
ACQ
USERS
REQ
SPEC
IMP
T&E
O&M
CIFO
EXPECTED CAPABILITIES
CAPABLE ITEMS
FORMAL COMPILERS/RTE/MODULES...
VENDORS/SUPPLY COMMUNITIES

Figure 4-2. Research and Development Issues.

The issues of access control in distributed environments were considered across a number of dimensions. As shown in Figure 4-3, the functional requirements of a project must be balanced against the constraints imposed on the solution (i.e., nonfunctional requirements). In turn, these issues must be balanced between the application software (responsible for the management of its own, unique services and resources) and the underlying system software (responsible for managing all services and resources shared across multiple applications and users).

	FUNC REQ	NON FUNC REQ
Ap Sw (Mng's Unique Serv. & Resources)		
Sys Sw (Mng's Shareable Serv. & Resources)		

Figure 4-3. Requirements Issues.

Figure 4-4 reflects the mapping of the Figure 4-3 concerns across a succession of target, integration, and host environments--each of which can be distributed. The semantics of access control in distributed target environments (where applications are deployed and operated) must be much richer than those of the current runtime environments if future projects are to satisfy their increasingly distributed and critical missions. In turn, this semantically rich, runtime environment of the target system will require enhanced support from the integration environment where final verification and validation of the target software is performed and the management of monitoring, advancing, and regressing the target environment baselines is performed. The combination of requirements for enriched runtime semantics among the target and integration environments impacts the requirements on the host environment where application solutions are proposed, developed, and sustained. Specifically, the ability to support dynamic, multilevel security and integrity in an incrementally evolving, distributed target environment requires access control semantics that are not found in today's systems and that must be developed and sustained in host environments and preserved across the integration environment.

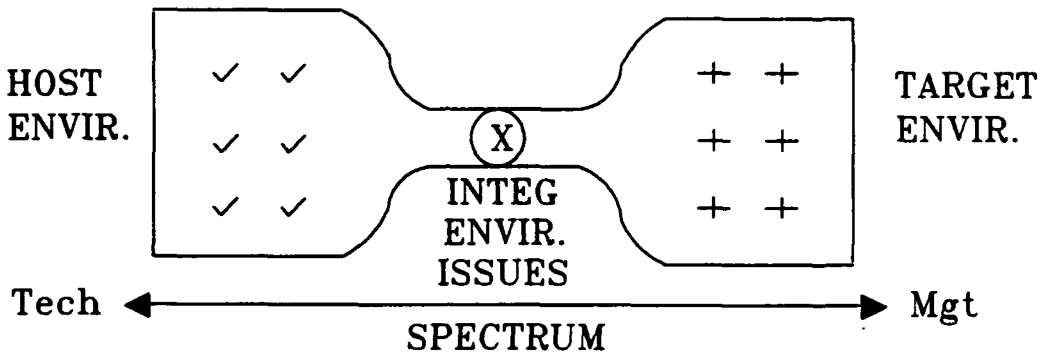


Figure 4-4. Environmental Issues.

The two parts of Figure 4-5 introduce the issues of a trusted computer base (TCB) which extends across portions of the hardware, the Ada runtime environment (as prescribed by the language standard), the extended runtime library (legal extensions such as those proposed in the ARTEWG CIFO*), and parts of the application. The right side of the figure extends these concepts to explicitly identify collections of processors, their individual kernels, and the supported applications. The TCB for such distributed systems would include the firewalled portions of the applications, supporting portions of the distributed kernel, and those individual processors and their kernels that are needed to support the dynamic, multilevel security and integrity (DMLSI) requirements of the applications.

*ARTEWG CIFO: Ada Run Time Environment Working Group - Catalog of Interface Features and Options.

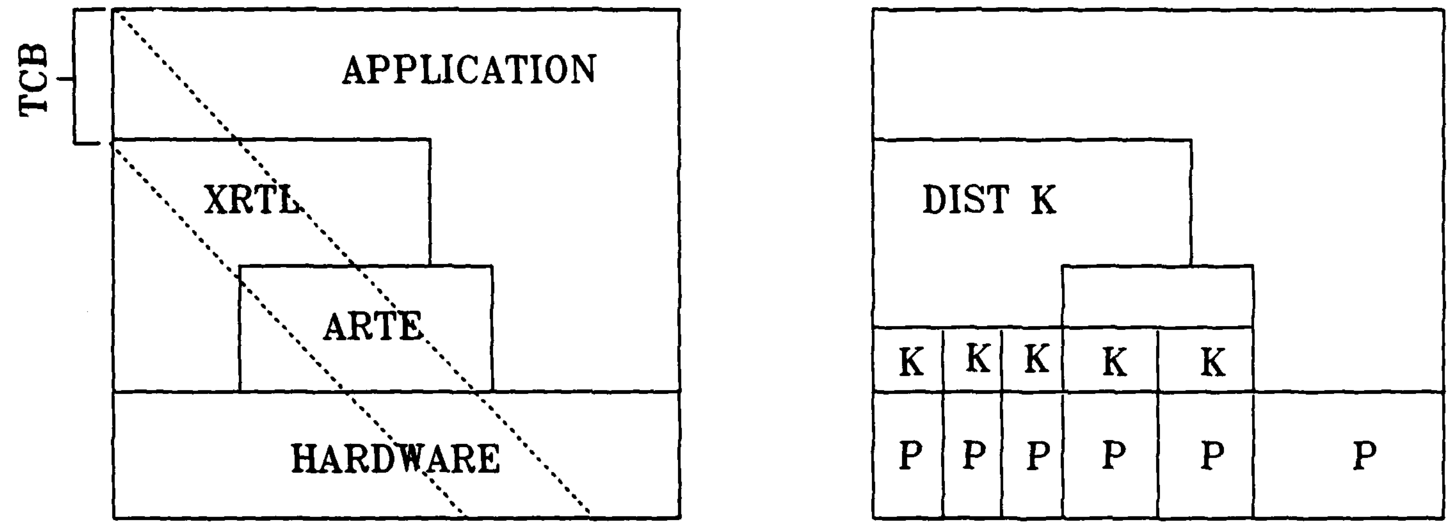


Figure 4-5. TCB Architecture.

The working group also considered the multidimensional issues involved in mapping the concerns depicted in Figures 4-1 through 4-4 to considerations of hardware, software, criticality and sensitivity, and time and space (see Figures 4-6 and 4-7). In particular, hardware considerations extend through the concerns of processors, buses, and the shared memory and devices of multiprocessor clusters to their interactions with other clusters via local area and wide area networks. The software considerations begin with the requirements for single processor kernels and extend through the multiprocessor and distributed kernels to distributed operating system libraries, configuration control, shared communications services and resources, shared information services and resources, and the distributed applications themselves. The concerns for criticality and sensitivity as well as temporal and spatial issues result in configurations which possess varying degrees of confidence and trust. In particular, the configurations reflect judgments of the system's logical components, physical components, and their mappings. Unfortunately, the research and development advancements which are sorely needed to resolve these issues cannot be reasonably expected to emerge from any one or two of the affected constituencies in government, industry, and academia. Instead, only a joint commitment is likely to have a chance to succeed in the face of so much complexity.

MAPPING CONSIDERATIONS

HARDWARE CONSIDERATIONS	SOFTWARE CONSIDERATIONS
WAN	DIST AP SOFTWARE
LAN	DIST INFO SERV & RES
MULTI PROC CLUSTERS	DIST COMM SERV & RES
PROCESSORS	DIST CONFIG CONTROL
BUSES	DIST RUNTIME/OS
SHARED MEMORY & DEVICES	LIBRARIES
	KERNEL, MULT PROC
	KERNEL, SINGLE PROC

FUNCTIONAL REQUIREMENTS & NON-FUNCTIONAL REQUIREMENTS
CRITICALITY & SENSITIVITY
TIME & SPACE

Figure 4-6. Hardware and Software Considerations.

CONFIDENCE & TRUST = F (ABOVE CFG'S

ABOVE CFG'S = F (LOGICAL COMPONENT AND PHYSICAL COMPONENTS
AND THEIR MAPPINGS)

AUDIENCE & PURPOSE

GOVERNMENT INDUSTRY ACADEMIA

COTS CUSTOM DEV RESEARCH

STANDARDS POLICIES

Figure 4-7. Other Considerations.

Figure 4-8 maps the preceding issues to the heart of the problem of DMLSI access controls in large, complex, distributed systems. Subject objects are processes representing application users in requesting the underlying system to provide the requested access to the designated target objects. The capabilities of these subject objects is intended to be a function of both the roles that may be assumed by the user and the views of the target objects that are permitted to the user in these roles. Often, the management of this subject object domain is separately vested from the management of the target object domain and the underlying distributed system of shared services and resources. Thus, the subject object with a given set of roles and views may prepare a requesting message for the destination site containing the target object. Assuming that the current context of the underlying system permits (e.g., no emergencies or overloads exist when the request is submitted by the subject object), the delivered message is checked in the target environment against the required access rights of the target object. If the capabilities of the subject match the required access rights of the target and the underlying system is prepared and able to support the requested access, then the request is honored.

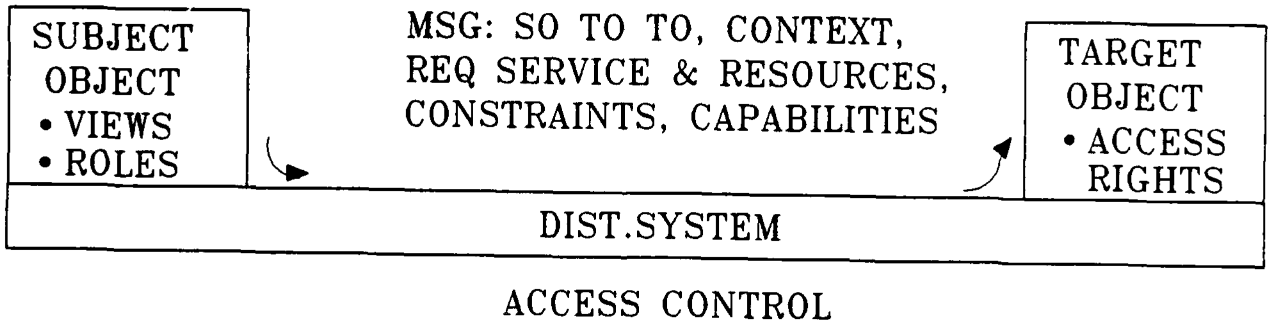


Figure 4-8. DMLSI Access Controls.

Figure 4-9 illustrates the mapping of the preceding concerns to the intended DMLSI environment. For example, distributed application 'A', part 1 of 3, might be the subject requesting access to a target object resource managed by the second of the three parts of this distributed application. When the subject object prepares and submits the requesting message to the underlying distributed kernel (i.e., the system software), it can be carefully monitored by background software that leverages the known semantics of the states and sequences of state transformations that are legal for the distributed application. Assuming no fault instance from a predetermined fault class was detected by this monitor, the message may be forwarded to the site of the intended target object. Here, the access rights required for the local target object are compared to the request prepared by the remote subject and the system software prepared meta information on the capabilities of the roles and views of the subject. A satisfactory match of subject capabilities, system context of operation (e.g., normal vs emergency), and target access rights may result in the satisfaction of the authorized-and-possible request.

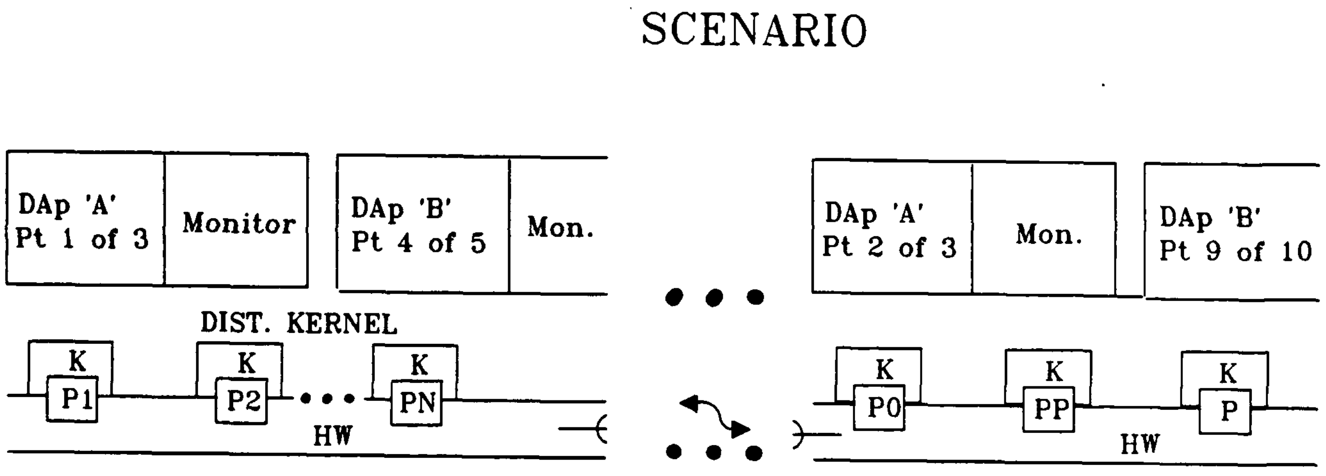


Figure 4-9. DMLSI Environments.

The following recommendations were developed by the working group in response to the preceding context and issues.

1. Evolve a national, Conceptual Reference Model (CRM) for runtime environments to support mission and safety critical applications in distributed environments.

The CRM should respond to at least the issues addressed in the preceding figures.

2.As the highest national priority for the use of the CRM, specify and develop the interface set of the distributed kernel.

The CRM interface set should support a 'single site image'.

Government contracts should follow for the development of proof-of-concept implementations, validation test suites, etc through formal models and methods for the distributed kernel and distributed applications.

3.Similar government contracts should follow for CIFO's (See the ARTEWG footnote.) of the: distributed information services, distributed communication services, distributed configuration control services, and distributed operating system services.

APPENDIX A

POSITION PAPERS

This appendix contains position papers from those workshop participants who agreed to have their position papers published in these proceedings.

Architectures for Secure Ada Runtime Support

Linda J. Harrison
Nina Lewis

Unisys Corporation
5151 Camino Ruiz
Camarillo, CA 93010

January 24, 1990

ABSTRACT

This paper addresses security issues related to Ada runtime support. Specifically, we examine architectural approaches to providing trusted runtime support, and the resulting requirements placed on the development environment. Two alternative approaches for runtime support are presented and examined: private Ada Runtime Systems, one for each application, and Shared Ada Runtime Systems, which are shared by several or all applications. Key security requirements, such as protection of the runtime library in the development environment and design of a security architecture in the operational environment, are examined. Both the Private and Shared approach alternatives seem feasible, but one may be preferred for a specific operational system.

1. Introduction

This paper reports on our investigation of security issues related to Ada runtime support, spanning both development and operational environments. Figure 1 shows how the Ada Runtime Library in the development environment is transformed into the Runtime System in the operational environment, and how the source program is transformed into a compiler-generated application program. Whether the operational system is able to protect its assets depends in part on the choice of protection mechanisms, but also on the correct implementation of those mechanisms. An appropriately chosen security policy for the development environment will increase assurance that the operational system mechanisms are implemented correctly. Thus, a key security issue in the development environment is protection of the Ada Runtime Library, so we examine the need to establish and maintain the integrity[1] of the Ada Runtime Library. A key security issue in the operational environment is designing the architecture of the virtual machine presented to the application.

The paper is organized in four sections. Since the security requirements of the operational environment should drive the security requirements of the development environment, we first present operational requirements in Section Two by examining how the Ada Runtime System can be accommodated within a trusted system that must provide security protection. Section Three then looks at requirements placed on the development environment by the operational requirements to support the development of Ada programs requiring runtime support. Conclusions are presented in Section Four.

[1] The type of integrity protection required is syntactic and functional, where syntactic integrity refers to constraining access without regard to an object's content, and functional integrity refers to the ability of a process to behave as expected.

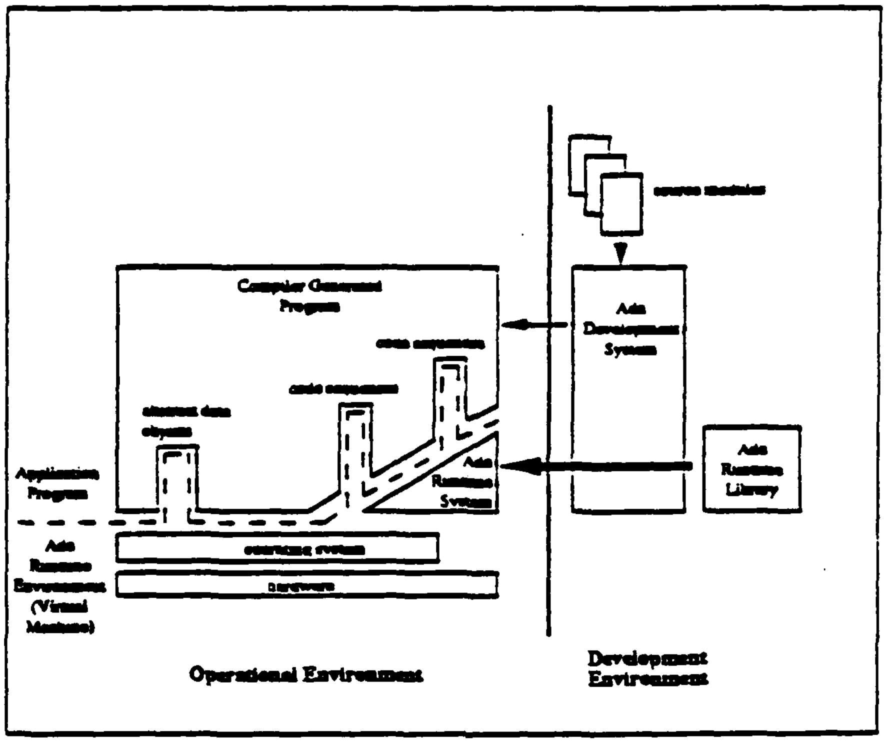


Figure 1 - Ada Runtime Support

2. The Operational Environment

This section describes various design alternatives for providing Ada runtime support. This runtime support may come from the underlying operating system, the Ada Runtime System, or a combination of the above. Using only underlying operating system support implies that certain Ada features (e.g. tasking) will not be used. Using only Ada Runtime System support implies that all capabilities not supplied by the hardware, but required to support Ada programs, will be provided by the Ada Runtime System. The combination approach implies that the Ada Runtime System will use services of an underlying operating system to provide services to Ada application programs. The application programs may also, if desired, directly use the services of the underlying operating system. This paper assumes that a combination approach will be used and that the underlying operating system is trusted.

2.1. Architecture of a Trusted System

Figure 2 illustrates a general architecture for trusted systems. At the lowest layer, a primitive kernel provides an interface to the hardware. Hardware memory management and interrupt processing are part of this layer. It is not until the extended kernel that familiar operating system abstractions appear. This layer implements the process abstraction and possibly file system abstractions. The security kernel implements the reference monitor concept, using the process abstractions of the extended kernel. In addition to the security kernel, there are trusted processes. The security policy allows trusted processes to violate the security policy of the security kernel, but only in a precisely controlled manner.[2] The process scheduler is a classic example of a trusted process. All of these items collectively make up an operating system TCB.

[2] While it is often stated that trusted processes are allowed to violate the security policy, formally the Bell and LaPadula security policy model handles this difficulty by considering trusted processes to be multilevel and permits a trusted process to write to any object within its range.

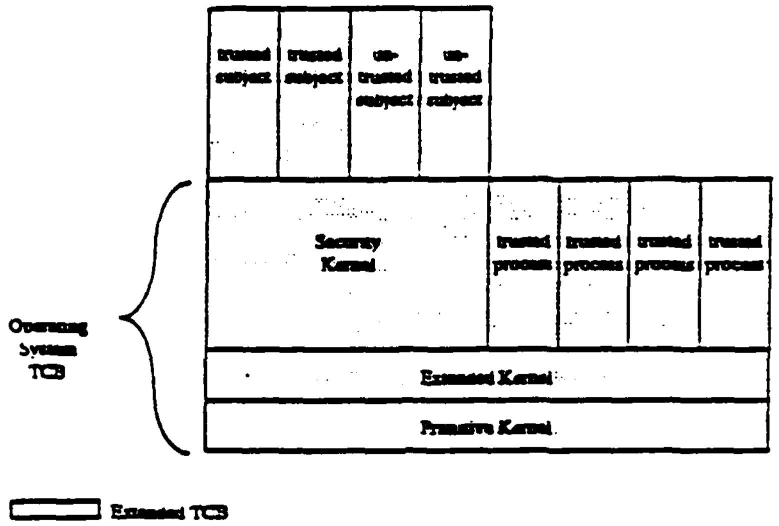


Figure 2 - Trusted System Architecture

Above the operating system TCB, all accesses to objects by subjects are mediated by the security kernel. Subjects are active entities that are implemented using the process abstractions of the extended kernel. Unlike trusted processes, they are labeled. There are two classes of subjects, trusted and untrusted. Trusted subjects also enforce a security policy[3], but their security policy cannot invalidate or interfere with the security policy of the operating system TCB (e.g. security kernel and trusted processes). Examples of trusted subjects are database management systems, mail handling systems, and simulators. Trusted subjects are very similar to trusted processes: the distinguishing factor is that the use of trusted subjects is mediated by the security kernel. The operating system TCB and all trusted subjects make up an extended TCB.

According to the Trusted Computer Security Evaluation Criteria (TCSEC), the TCB is the totality of protection mechanisms that enforce a security policy, and trusted software is the software portion of the TCB. Yet, it is not clear if the protection mechanisms for integrity and service assurance policies can be implemented within a TCB framework. That is, the protection mechanisms for integrity and service assurance may be distributed throughout the system, making it impossible to distinguish TCB and non-TCB elements. In this paper, we consider trusted software to be the software portion of a TCB, but recognize that there may be other software that is security relevant but is not part of a TCB.

2.2. Private Runtime Systems

One proposal regarding Ada runtime support in a trusted environment is a structured approach with four components: untrusted applications, untrusted runtime systems, the trusted security kernel, and the trusted security kernel runtime system. Each application and the kernel maintain private runtime systems. The security kernel's runtime system is architecturally part of the security kernel and thus within the operating system TCB. The application runtime systems are architecturally part of the applications. They cannot subvert the protection measures of the kernel and thus need not be trusted. This proposal is somewhat simplified. It does not consider the architecture of a trusted system as presented in Figure 2. The operating system actually consists of a layered kernel and trusted processes. There may be a runtime system associated with all of these entities, as well as a runtime system associated with each of the system's applications. Further, some of the applications may need to be trusted (e.g. trusted subjects).

[3] Another term for trusted subject is trusted application.

While it is hard to predict whether some applications can be developed without using Ada features that require runtime support, there is evidence that a trusted operating system can be developed in Ada without using runtime support, namely the Army Secure Operating System (ASOS). Therefore, in the remainder of this paper, we assume that the trusted operating system components require no runtime support, and concentrate on how to provide runtime support to application programs.

In Figure 3, an architecture is proposed that shows private Ada Runtime Systems for each application. The runtime systems of trusted subjects must be trusted: their design and development must be subjected to TCSEC requirements. If this were not the case, malicious code could subvert the security policy of the trusted subject. The runtime systems of untrusted subjects are not trusted, because they necessarily cannot subvert the protection measures of the extended TCB. Security relevant features of the Ada Runtime System are only security relevant if the runtime system is required to handle multiple levels of data. The private runtime systems of untrusted applications will not handle multiple levels of data.

A shortfall of this proposal is that it assumes that the security policy is enforceable by a centralized TCB. While a confidentiality policy can be enforced in this manner, it is not clear that integrity and service assurance requirements can be enforced within a small centralized TCB. Thus, there may be integrity and service assurance requirements placed upon applications that are not part of a TCB. For example, it may not be desirable to introduce unknown or unscrutinized code into any application with integrity or assurance of service requirements. This implies that Ada Runtime Library routines supplied by vendors in executable form may not be introduced into the Ada Runtime System for trusted applications. We return to the issue of functional integrity of Ada Runtime Library routines during our discussion of the development environment.

Building a secure application requires considerable effort during all phases of the software lifecycle. With this approach each application is considered separately, and the process of evaluating the application and the application runtime system is performed individually for each application. Using runtime support will make the application larger and thus more difficult to evaluate. This could result in a tendency not to use Ada features, such as tasking, using instead system level routines. The result - less portable applications.

2.3. Shared Runtime Systems

A logical alternative to private runtime systems are runtime systems that are shared by several or all applications. We first considered proposing an Ada Runtime System as a trusted process in the operating system TCB, but the TCSEC is very specific about what can and cannot be part of the operating system TCB. Beginning with class B2, the TCB must separate those elements that are protection critical from those elements that are not. Class B3 and above requires that the TCB should be minimized in complexity and *may not* contain modules that are not protection critical. Protection critical elements are those whose normal function is to deal with the control of accesses between subjects and objects. Clearly, there are both features that are and are not protection critical for a runtime system providing services to several applications.

There are two fairly convincing reasons to argue that the runtime system shouldn't be a trusted process in the operating system TCB. First, since trusted processes are not mediated via the security kernel but are part of the TCB, they are subjected to stringent assurance requirements. In particular, at levels B2 and above a thorough covert channel analysis must be performed. Second, it is hoped that in the future many secure operating systems will become available as Commercial Off The Shelf (COTS) software. To expect that such systems will include an Ada Runtime System as part of their TCB is unreasonable, because not all customers will want such support.

Rather than reject the concept of shared runtime systems, we propose that shared runtime support be provided via a trusted subject (see Figure 4). With this proposal, a shared runtime system, which is architecturally an operating system subject, supports several applications. Careful consideration is needed to determine if all applications will be supported by the same runtime system. Possible scenarios include:

- One shared runtime system.
- One shared runtime system to support all trusted applications, and one shared runtime system to support all untrusted applications.

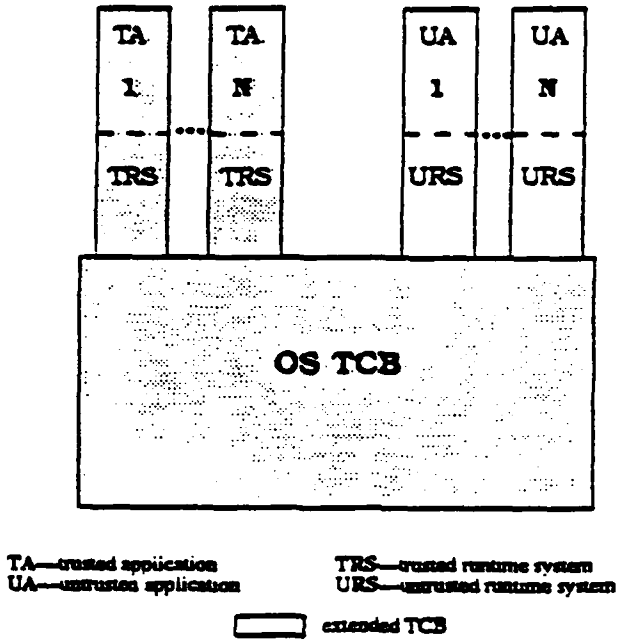


Figure 3 - Private Runtime Systems

• All applications at a given security level share the same runtime system.

If the runtime system is shared among applications at different security levels, it will have to be trusted. An untrusted shared runtime system is feasible if sharing is restricted to single, same-level applications. However, even if the shared runtime system is untrusted, its functional integrity (its ability to behave as expected) should be evaluated.

One disadvantage to this proposal is performance. Each application that uses the shared runtime system requires a minimum of two context switches when runtime support is requested. The first context switch to make the application program inactive and the shared runtime system active, and a second context switch to make the shared runtime system inactive, and the application program active. All context switches are mediated by the security kernel, which will also slow things down. Shared memory might reduce the number of context switches required in some instances.

2.4. B2-Like Requirements, Shared Runtime Systems

Before a Trusted Ada Runtime System can be built, the criteria for building one must be understood. In this section, we look at TCSEC criteria and begin interpretation of the B2 criteria for a Trusted Ada Runtime System. This interpretation applies specifically to shared runtime support, because the criteria for private runtime support must be interpreted within the domain of the individual application.

2.4.1. Security Policy

A security policy is a statement of rules, laws, and practices that regulate how information is managed, protected, and disseminated. Since the TCSEC mainly addresses confidentiality (as opposed to integrity and service assurance), this interpretation of the TCSEC will do likewise.

A security policy and formal security policy model for the Ada Runtime System will need to be developed. Bell and LaPadula, the most often used formal model, requires identifying subjects and objects and then showing that access to objects by subjects does not violate the simple security property or the *-property. The only subjects will be Ada programs. The objects will probably be the same as operating

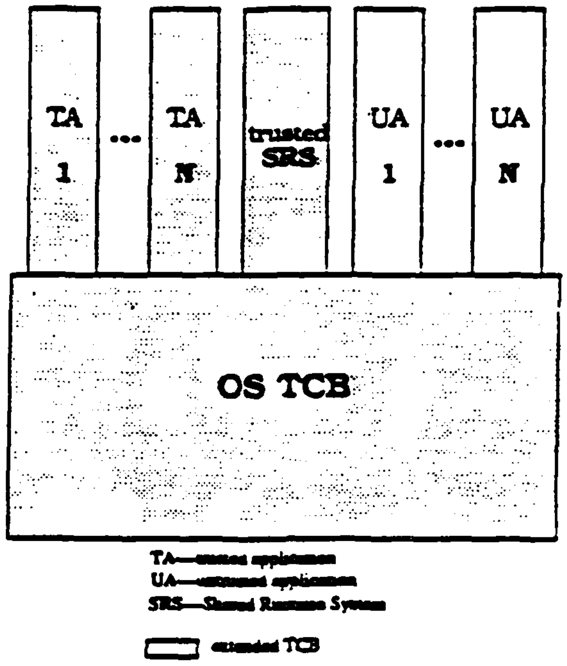


Figure 4 - Shared Runtime Systems

system objects (e.g. files, devices and memory).

2.4.2. Accountability

There must be a means to assure individual accountability of actions. Thus it must be possible to authenticate an individual's identity and audit an individual's actions. The TCSEC identifies the need for passwords to authenticate individual users. The only users of the Ada Runtime System will be Ada programs and their identity should be authenticated by system labels. Audit requirements should be very similar to those currently raised in the TCSEC.

2.4.3. Assurance

Assurance objectives help to guarantee that the security policy is enforced. Lifecycle assurance requirements ensure that the system was designed, developed, and maintained properly. Operational assurance requirements ensure that the system architecture provides protection from external interference. We have identified several issues that must be addressed.

1. *All interfaces between the Ada program and the Ada Runtime System must be defined.*

   To satisfy B2 criteria, the TCSEC requires that a descriptive top level specification (DTLS) of the TCB be developed and maintained. It must be an accurate description of the TCB interface.[4] ARTEWG has done some preliminary work in this area by defining "A Model Runtime System Interface for Ada." This work could be used as a starting point, but the actual TCB interface will probably be somewhat different.

2. *All interfaces between the Ada Runtime System and the trusted operating system must be defined.*

[4] Note that this is actually the extended TCB interface (see Figure 2).

It is assumed that there will be a trusted operating system (either Ada or non-Ada) underlying the Ada Runtime System. The DTLS for the Ada Runtime System must identify all interfaces to the underlying operating system. This is in addition to the requirement that all interfaces between the extended TCB and the Ada program be identified.

Note that this requirement is not identified in the TCSEC. The TCSEC was developed for general purpose operating systems. It does not specifically recognize (although it similarly does not prohibit) the trusted system architecture presented in Figure 2. We believe that all trusted subjects will have to identify two TCB interfaces: between the extended TCB and user, and between the trusted subject and operating system TCB.

3. *A thorough search must be conducted to ensure that covert channels are not introduced.*

The trusted operating system will enforce mandatory access control, but the introduction of a trusted Ada Runtime System could introduce covert channels. Recall that there are many alternatives for the runtime system design, ranging from all Ada programs sharing one runtime system to only programs of the same security level sharing runtime systems. If any alternative is chosen that allows Ada programs with different security levels to share runtime systems, then covert channels could be introduced and must be analyzed.

This does not represent a complete interpretation of the TCSEC for Trusted Ada Runtime Systems, but it does address some of the most obvious and interesting requirements that need to be considered in developing a B2-Like Ada Runtime System. Integrity and service assurance requirements must also be defined.

3. The Development Environment

Thus far, this paper has been concerned with the security issues of providing runtime support in the operational environment. It has identified two alternatives for providing such support: 1) a private Ada Runtime System that is included as part of each Ada application program; and 2) an Ada Runtime System that is shared by several (or all) Ada application programs. Now we examine how the security requirements for each of these alternatives are related to the security requirements for the development environment.

3.1. Private Runtime Support

In this section, the provision of separate runtime support for each application is examined. As Figure 5 shows, there is an Ada Development System (ADS) that consists of several software tools that perform the task of transforming Ada source programs into Ada executable programs. The inputs to the ADS are sources modules and Ada Runtime Library routines. The output is an executable module, which may represent either a trusted or untrusted Ada application. The following discussion identifies requirements placed on the development environment for trusted applications (those applications that are part of a TCB), and, where appropriate, requirements for untrusted applications.

3.1.1. Assumptions

The quality of the source modules is one of several factors that can affect the executable module produced in the development environment. Nonetheless, for the purposes of this paper, we assume that the functional integrity of the source modules is beyond refute.

3.1.2. Functional Integrity of Runtime Library Routines

A key concern is the introduction of runtime library routines into trusted applications. The Ada Runtime Library is a set of routines, typically not available in source form, provided by the ADS vendor. Introducing these routines, which potentially contain malicious code, into a trusted application is a violation of TCSEC assurance requirements. Malicious code placed in a trusted application in the development environment could result in a security policy violation in the operational environment.

Thus, it must be demonstrated that the Ada Runtime Library routines have functional integrity. Source of the routines must be available, and the routines must be shown to be correct, complete, and exact implementations of their requirements. Though we recognize that it is not possible to show absolute

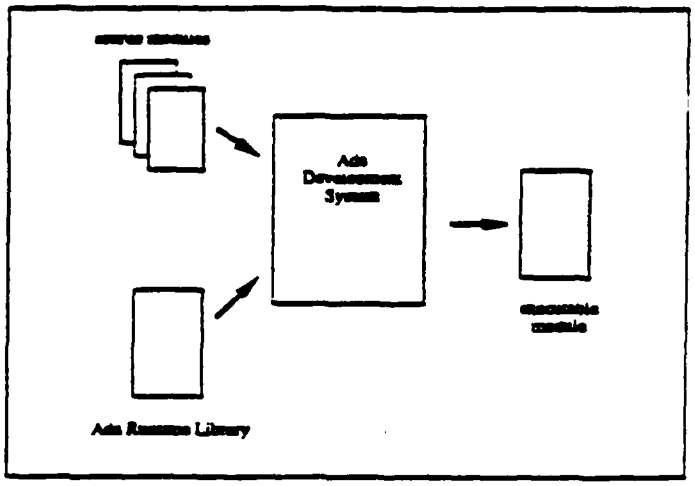


Figure 5 - The Development Environment

functional integrity, appropriate assurance techniques for the level of trust desired must be used. These may include: formal specifications, informal specifications, testing, and extensive peer review.

Functional integrity of Ada Runtime Library routines may also need to be established for untrusted applications. As described earlier, we define a trusted application as enforcing a security policy within a TCB framework. There may be applications that contribute to overall integrity and assurance of service requirements, but that are not part of a TCB. We expect that these untrusted applications shall also require the Ada Runtime Library routines to have functional integrity.

3.1.3. Syntactic Integrity of Runtime Library Routines

Once the functional integrity of Ada Runtime Library routines is established, it must be maintained. Thus, there must be implemented within the development environment an integrity policy that enforces the syntactic notion of integrity. Syntactic integrity constrains access purely on a computational level, without regard for the content of the protected resource. The syntactic integrity policy must prevent unwanted modifications to Ada Runtime Library routines. Biba defined a mandatory access control policy to provide such protection, but a discretionary access control policy might also be used.

3.1.4. Compiler Interactions

There is a close tie between Ada runtime library routines and the Ada compilation system. During compilation, the compiler translates the Ada program into machine language. The compiler has two choices for providing runtime support. The compiler may provide runtime support by implicitly invoking runtime routines, or the compiler may simply generate inline code. The runtime routines are said to be invoked implicitly because calls to supporting routines are not visible in the source code. [5]

The preference is to have runtime support provided with implicit calls to runtime library routines, rather than inline generated code. This approach allows a clear delineation between the Ada Runtime Library and the Ada Language Compiler, beneficial because it is significantly more difficult to provide a trusted Ada compiler than it is to provide a Trusted Ada Runtime System. By carefully delineating the two tasks, a Trusted Ada Runtime System can be built without a trusted Ada compiler. Of course, we must recognize that the untrusted compiler may sabotage the runtime system.

[5] Ada also uses explicit runtime support, where calls to Ada Runtime Library routines are visible in the source code. Compiler interaction with explicit runtime support is less of a concern because it merely requires that the compiler correctly generates code to invoke a subroutine.

3.2. Shared Runtime Support

When shared runtime support is being provided for applications, the development environment changes. The ADS still performs the function of transforming an Ada program into an executable module, but the Ada Runtime Library routines are no longer an input to the process. The runtime routines have been incorporated into an executable module of their own, and are already installed in the operational environment as the Ada Runtime System. Functional integrity of the runtime library routines and syntactic integrity protection are issues that must have been considered when the shared Ada Runtime System was built. Similarly, how the compiler and the runtime library routines divide the responsibility of providing runtime support must have been defined when the shared Ada Runtime System was built. There are no Ada specific responsibilities placed upon the development environment when shared Ada Runtime Systems are used.

4. Conclusions

In this paper, we have examined security issues related to the runtime support that is required of Ada programs. Specifically, we examined architectural approaches to providing trusted runtime support, and the resulting requirements placed on the development environment. Two alternative approaches for runtime support are presented: private Ada Runtime Systems, one for each application, and Shared Ada Runtime Systems, which are shared by several or all applications. Both alternatives seem feasible, but one may be preferred for a specific operational system.

The next step is to begin to build a trusted Ada Runtime System, by developing a detailed but informal specification of the security relevant Ada Runtime Library routines. This specification would serve as the basis for a DTLS for shared runtime support and as documentation of runtime library routines introduced into the application for private runtime support. The ARTEWG Model Runtime System Interface can be used as a starting point for this work.

POSITION PAPER

ACCESS CONTROL FOR A SAFETY CRITICAL DISTRIBUTED SYSTEM INTERFACE SET

Sue LeGrand. Planning Research Corp.

January 2, 1990

The Problem

The National Aeronautics and Space Administration (NASA) must guarantee the integrity of mission and safety critical components in large, complex. non-stop, distributed environments such as the Space Station Program, the Lunar Base, Mars exploration. and other projects for which Ada is the programming language of choice. Run time issues include real time performance, reliability, fault tolerance, survivability, and dynamic extensibility and reconfiguration in non-stop environments.

All access to safety and mission critical components of the system that is granted to users must be monitored and controlled for the appropriate operations, resources and constraint enforcement of time, location and other restrictions. In addition, access may be dependent upon modes of operation; different management domains for subjects, targets and intervening message paths; and normal versus exception contexts. Integrity must be assured at all times in the life cycle against the hacker who inadvertently causes damage, the terrorist who maliciously seeks to cause destruction, or a disgruntled former staff member who leaves with potentially dangerous knowledge. Furthermore, there must never exist a thread of control that runs amok (i.e., may be unresponsive to commands and is in an eternal loop occupying system resources and/or may access critical resources).

Systems which contain only the policies, manual procedures, encryption, passwords and other traditional means of security and integrity management cannot provide solutions to the above challenges. Much of the public domain research to support computer security and integrity has been conducted in a static host environment that uses shared services and resources managed at a low level with an untyped interface. These systems are a composite of different paradigms and device characteristics.

The preoccupation with these static issues does not provide solutions to the dynamic (runtime) issues of a big and growing class of applications; namely large, complex, non-stop, distributed systems which evolve incrementally and must provide life cycle support for mission and safety critical components.

A system is needed with a runtime enforcement of the security and integrity policies. In order to provide this, three needed elements are:

- A strongly-typed object oriented operating system with assertions-based enforcement of constraints.
- A dynamic, capability-based addressing design for subjects.
- A dynamic access control list design for targets.

The Prototype

Sponsorship is being sought for a security and integrity assured prototype to be developed within the context of a mission and safety critical system that supports fault tolerant and fault recovery software and hardware components.

The models, techniques and tools will be developed to support the goal of runtime software instruments for automatically evaluating the security and integrity effectiveness of object based computer systems.

The models, techniques and tools will be designed and evaluated in a *multiuser, multilevel security* and *integrity* environment that is supported by an object management system with assertions-based enforcement of constraints in the system software. This object management system will control access to all resources of the system, rather than just the database. This environment will be used to demonstrate optimal performance and ease of use with a transparent, distributed and multiprocessing computer system.

This research will demonstrate the benefits of using consistent and precise models based on an object oriented paradigm to satisfy the security and integrity requirements of computer systems. This research will be conducted within the context of a mission and safety critical distributed, non-stop, multicomputer system that must satisfy dynamic access control of multiple users, even in the presence of faults, upgrades and reconfigurations.

A plan for testing this type of access control calls for multiprocessor clusters (at least 3 processors per cluster) interconnected by a high speed, deterministic local area network (LAN). In turn the LANs are connected by a wide area network (WAN). The plan calls for a test bed WAN to integrate a minimum of 3 LANs where each LAN has a minimum of 3 clusters and each cluster has a minimum of 3 processors.

An associated goal is to evolve the testbed to demonstrate proof-of-concept of dynamic extensibility such that no software changes are needed to add or remove LANs, add or remove clusters of a LAN, or add or remove processors within the cluster. The systems software (e.g., the processor kernels and the run time library modules) provide support for the access control while also satisfying the requirements listed above. This includes non-stop modifications to advance or regress the operational baseline.

This prototype project is a subset of a larger effort of the High Technologies Laboratory (HTL) located at the University of Houston - Clear Lake near Houston, Texas. The HTL goal is to reduce risk in computer systems and software engineering issues which are considered critical to future missions of NASA. HTL works with private industries and government agencies other than NASA. The project is associated with the Software Engineering Research Center (SERC), which is a wholly sponsored research center by the long term research division of NASA Headquarters with the same mission and priorities as HTL. Dr. Charles McKay is director of both SERC and HTL.

Low Ada and a Trusted Ada Kernel

John McHugh

Computational Logic, Inc.

30 January 1990

One of the potential worries in developing trusted Ada applications is the trustworthiness of the Ada runtime system. Anyone who has been confronted with a half megabyte core image of a trivial program has wondered,

1. "Is this really necessary?"

2. "What's *in* there?"

Part of the problem comes from the immaturity of the Ada technology, especially as regards partial use of predefined and library packages. Another part of the problem comes from a tendency of developers to simplify compilation through the extensive use of runtime support.

The results of such an approach run counter to the minimalist maxims of trusted systems. TCBs that are intended to provide high levels of assurance are constrained to contain all and only the trust enforcement mechanisms of the system in question. This is true whether the system in question is being used to provide security in the sense that information compromise is prevented, or whether it is being used to assure the safe operation of a vehicle or other machine. The minimality of the TCB is undermined if the size of the runtime system required to support it vastly exceeds the size of the code due to the compilation of the TCB.

Several recent developments hold promise for helping to bring this situation under control. Requirements from the Ada 9X workshop held last spring in Destin, Florida call for compiler vendors to disclose more of their implementation decisions. This includes decisions that affect the choice between runtime support and inline code for certain constructs. This call was echoed at a recent Ada 9X trusted systems workshop hosted by IDA, especially with respect to heap management and the need to avoid the use of a heap for well-behaved applications that eschew explicit dynamic allocation.

The latter workshop also brought forth a call for both a standard intermediate language (Brian Wichmann's Low Ada proposal), and for the development of a standard runtime kernel for use with trusted or safety-critical systems. Low Ada is viewed as an Ada-like language with a simple static semantics that can be produced from an Ada front end. Low Ada would have formally-defined static and dynamic semantics that would permit the verification of Low Ada programs as well as other forms of static analysis. The trusted kernel can be viewed as providing runtime support for, at least a subset of, the program that could be written in Low Ada.

I propose an extension and unification of these ideas. Low Ada is a much more flexible language than Ada. Many of Ada's type restrictions and enforcement mechanisms would be enforced in the Ada to Low Ada translation process, but are not part of the Low Ada semantics. This makes it possible to write many programs simply in Low Ada that are difficult or impossible in Ada. Substantial portions of a runtime support system fall into this category. Although originally intended as an intermediate language alone, it appears that Low Ada could be used to implement large portions of an Ada runtime system and/or an operating system kernel for use in support of trusted and safety-critical applications. Given a formal basis for Low Ada, a formal specification for such a kernel would appear to be tractable.

Development of such a kernel would provide an opportunity to explore a number of trade-offs that concern both security and integrity. If heap management is included, how is memory reuse achieved? Are there reasonable ways to support multilevel tasking in Ada? What is the role of the runtime system in promoting data integrity? Can such a system provide process integrity?

It is clear that this proposal presents a number of research issues, some of which are already being addressed in other forums. Work at Computational Logic, Inc. is intended to produce a formally-defined subset of Ada, known as AVA, with a rigorous definition in Boyer-Moore logic. An attempt to provide a formal translation mechanism from AVA to Low Ada is being considered. Issues connected with the formal specification of operating systems are being addressed in the context of Mach by several researchers. Specification components for Ada are available in the form of Anna and Larch. Work to link "Z" to Ada is being considered.

A trustworthy Ada runtime kernel in Low Ada would draw on all these efforts, and could provide a vehicle for unifying and focusing Ada trusted systems research in the near term. As Ada approaches maturity, it is important to look beyond the problems of first generation systems and address some of the open issues in security and integrity in an Ada context.

Identity as a Basis for Ada Run Time Environment Security and Integrity

Jon Shultis

February 16, 1990

[1]

1 Position

Extension of the language-level policy of strong typing to all resources in an Ada environment, coupled with more expressive typing, provides an effective and practical means of achieving security and integrity.

2 Background

For present purposes, a *security breach* is any behavior of a system that provides unauthorized access to information or control. It does not matter whether the errant behavior is produced intentionally or accidentally; in either case there is an opportunity for abuse. The phrase "any behavior" is here meant in its broadest sense, so that such things as covert timing channels are covered by this definition.

An *integrity breach* is any modification of information or control which compromises its accuracy. It matters not whether the modification is authorized or unauthorized; in either case improper computations, decisions and actions may result. Obviously, integrity breaches may compromise security by corrupting mechanisms that enforce policies, and *vice versa*.

Note that these definitions classify accidental modifications of data due to environmental conditions as integrity breaches. Such breaches of integrity are to some extent unavoidable, and more robust systems will enable detection and repair of integrity failures. In what follows, however, we

[1]This work was supported in part by DARPA under contract MDA 972-88-C-0076.

shall focus on the problem of maintaining security and integrity in an idealized world of flawless physical devices.

The problem before us is to find a means of achieving security in Ada run time environments. The solution we propose is based on a familiar, if somewhat naive, view of strong typing disciplines as language-level policies for controlling access to information and control.

In this view, any operation in a program is regarded as a consumer and producer of resources. The signature of the operation specifies the type and access capabilities which the operation has for each resource it manipulates. Language rules for inheritance and subtyping govern the propagation of access authorizations. In "orange book" terms, we have the following rough correspondence:

- type discipline = security policy
- typing = marking
- overload resolution (operator identification) = identification
- type checking and error reporting = accountability
- compiler validation = assurance
- compiler protection = *continuous protection*

With this correspondence, a "security breach" occurs in an Ada program if an operation is permitted to access data of a type or in a mode not authorized by its signature.

The specific policies of the Ada type discipline do not fulfill many of the specific objectives for various models of trusted systems, such as hierarchical sensitivity classification. Moreover, the responsibility for enforcement even of the type discipline does not extend very far beyond the boundaries of a compilation unit. We contend, however, that an appropriate generalization and extension of the type discipline to all resources in the environment can be used to fulfill the objectives of trusted systems at a very high level.

Two independent extensions are indicated. The first is to extend the domain of the type discipline to include all resources in the environment. Current Ada run time environments assume that any responsibility, *even for simple Ada type integrity*, stops at the boundary with the host operating environment. For example, when a file with a given Ada type is written, it is represented by an object in the host environment which carries no information about its Ada type, and its integrity as an object of that type is not guaranteed by the host environment. From a security standpoint, this is intolerable; it is simply not possible for a system to be secure if it routinely transfers information of all kinds to an insecure domain where it may be subjected to arbitrary analysis and transformation without any enforceable controls or audits. Of course, this problem is not unique to Ada, but pervades the architecture of all compartmentalized, as opposed to integrated, software systems (which is to say, all existing commercially available systems).

The second is to extend the expressiveness of types to enable specification of properties of information and control which are relevant to security, such as resource behavior and sensitivity classifications. Note that the extended type system which we propose is to be imposed on the run time environment, but not necessarily on the Ada language itself. In order to be used in a secure environment, components written in Ada would, however, be required to undergo an analysis to verify their properties *vis a vis* the security requirements of the intended application. Thus our proposal can be realized either by changes to Ada, or more conservatively by requiring Ada components to obtain security clearance before accessing sensitive resources. Specification of security properties and analysis of programs relative to those specifications can be achieved in a variety of ways, for example along the lines of Anna.

The remaining discussion is devoted to explaining what extensions are required and how they can be accomplished.

3 Strongly Typed Environments

Any access control policy which is to be enforced throughout a run time environment requires positive *identification* and *classification* of all resources in the environment, including persistent data, processes, and (physical as well as abstract) channels. Positive identification requires only that every (recognized) resource be assigned a universally unique, location-independent identifier (uid). Positive classification requires that each resource be associated with a type. Note that every type is itself a resource, so the association can be represented by a pair of uid's, one for the resource and one for its type.

A remarkably simple notion of type suffices to obtain the expressiveness required for classification: a type is a set of properties, closed under entailment. Thus, for example, one property of a type of interactive program might be that its response times are independent of memory and cpu loads, at least within a given range. This is not the place to expand on this topic, but it is worth noting that there is a natural notion of subtyping and inheritance in such a type system, in which subtyping amounts to specialization. This feature is critical to the definition of such things as security classes, which are metatypes (types of types), the examples of which are types that carry security properties common to the class. Access controls are defined on classes, and automatically inherited in a uniform way to all subtypes. Such things as hierarchical classification lattices can be embedded in the type lattice in the obvious way.

Of course, determining whether a given Ada program exhibits such properties is dependent on the state of verification technology, and we fully expect that many types which would be used in specifying secure systems would be extremely difficult to verify with current technology. This should not prevent us from specifying them, however, nor should it prevent us from providing provisional, or partial, assurances. A system in use is suspect to the extent its clearances depends on incomplete proof obligations; but it is useful to know precisely what the expectations for a system are, and the extent to which those expectations are not known to be met.

The library system for an Ada compiler may provide some insight into how these extensions coöperate to implement policy in the environment in a relatively simple situation. One of the more enlightened aspects of the Ada design is its recognition of the need to extend the domain of responsibility of the type system beyond the boundaries of individual compilation units. Ada implementations are therefore required to enforce a policy governing access to library units, where in this case the subjects are library units and the objects are other library units named in context clauses. By associating a universal id with each compilation of a specification, it is possible to assign a "signature" to each library unit. The signature specifies the other library units which the given unit is authorized to access. The rules regarding compilation order, access to library units and subunits, and linking amount to an access control policy governed by these signatures.

When a unit is compiled, symbolic references to other units are resolved to corresponding universal id's, and the symbolic names are discarded. In all subsequent uses of the compiled unit, its references to other library units are mediated by the universal id's.

When a unit with a given symbolic name is recompiled, the objects which previously went by that name in the library are replaced, and become inaccessible. Thus any subsequent attempt to use a unit containing a reference to the obsolete unit will fail, thereby enforcing Ada's policy on compilation order.

For another simple (but perhaps more pertinent) example, let c be a security class, and let c(file of t) designate the subtype of class c consisting of files containing objects of type t. Access to such a file is mediated by the run time environment, which must certify that the agent accessing the file has the authority to do so. Suppose that the immediate agent is an Ada program, and the authority required is that the program has certified the authority of a human user interacting with the program to obtain the given access. The security property which is required of the program is that it has carried out some authentication protocol with the user prior to requesting the access. If the program in question has been certified (by whatever means) as always carrying out the required protocol, then that fact will have been recorded in its type (signature). It is therefore a simple matter for the run time environment to clear the program for the requested access by matching the relevant part of its signature against the type of the classified resource.

USING ADA TO SIMPLIFY ENCRYPTION SYSTEM ARCHITECTURES

William R. Worger, *Senior COMSEC Systems Engineer*
Michael A. Diaz, *Senior COMSEC Software Engineer*

ABSTRACT

Traditionally, guaranteeing that Red (sensitive) data is not accidentally released to an adversary has been one of the most difficult and expensive problems in designing encryption systems.

The difficulty arises in attempting to prove that Red data cannot be released to the adversary even in the event of a hardware failure. Typically this proof takes the form of a Single Fault Analysis (SFA) on the system. The SFA analysis is used to verify that any single failure (Hardware or Software) will not cause a release of sensitive data to the adversary. The cost of this SFA analysis increases exponentially with the increased complexity of the hardware or software. To reduce the expense of this analysis, the system designer must partition the hardware and software into small, easily analyzed sections.

Traditionally, encryption system architects completely separate the portion of the system which processes Red information from the portion of the system which performs black (non-sensitive) information. This results in a system which contains two processor subsystems; a Red processor subsystem and a Black processor subsystem.

From a performance standpoint, the dual processing system is not necessary since today's processor can easily handle both the Red and Black processing loads in the encryption system. The result is that traditional encryption systems contain twice as much processing circuitry as would be required if only one processor were used to perform both the Red and Black processing functions.

This paper presents a single processor encryption system architecture which uses various features of Ada to ensure separation of Red and Black data in the system. Also discussed are various precautions which must be employed when using a high level language in an encryption system. This paper concentrates on the software architecture of the system but also includes several hardware considerations.

INTRODUCTION

In traditional encryption systems, guaranteeing that Red (sensitive) data is not accidentally released to an adversary has been one of the most difficult and expensive problems facing the designer of these systems. It is generally regarded that the only thing worse than not encrypting sensitive information is thinking that the information is being encrypted correctly when in reality it is not. In this situation the user freely transfers sensitive information because he or she trusts that the encryption system is protecting the information properly.

To guarantee that the system will not reveal sensitive information, even in the event of a failure, the system designers will generally perform a Security Failure Analysis (SFA) on the system. The SFA analysis attempts to prove that the system cannot fail in a way that would allow sensitive information to be released to an adversary. The SFA analysis should cover both the hardware design and the software design to be effective.

The cost of proving the correctness of both software and hardware increases exponentially with the increase in the complexity of the system. This has forced the system designers to develop architectures that attempt to isolate the Red data from the Black data.

Figure 1 shows a typical architecture which reduces the SFA effort. The architecture is separated into two isolated subsystems: the Red processing subsystem and the Black processing subsystem. The Red subsystem preforms all of the processing and memory management associated with the sensitive data and the cryptographic keys. The Black subsystem performs all of the processing of the encrypted data and usually performs the user interface functions. Each of these subsystems contain separate RAM, ROM, microprocessors, clocks, and address decoding circuitry. The Red and Black subsystems are connected via some type of encryption system which encrypts all data being transferred from the Red subsystem to the Black subsystem.

This architecture is very easy to analyze since all data which flows from the Red subsystem to the Black subsystem flows through the encryption system. To prove that Red information is not released to the Black processor, only monitoring the encryption block for correct operation is needed. This can be done by using redundant encryption blocks or by placing monitors to verify the operation of the encryption system.

Although the architecture in Figure 1 is ideal from an SFA standpoint, it is very inefficient from a hardware and software usage standpoint. In many systems, a single processor has sufficient power to handle all of the processing needs of the entire system. In these cases the architecture in Figure 1 will be twice as expensive and take twice the room of a system which shares a single processor to preform both the Red and Black processing functions. The system in Figure 1 will generally require two operating systems, two memory management system and two sets of self-test software, one for each processor.

In the remainder of this paper we will present an alternate architecture which shares one processor to perform both the Red and Black data processing. We will discuss how the Ada language can be utilized to simplify the SFA analysis of the system and guarantee Red and Black data separation.

The proposed architecture is shown in Figure 2. This architecture is similar to the traditional architecture in that there are separate Red and Black memories which are connected by an encryption function. The architecture deviates from the traditional architecture in that the system uses the same processor to perform both the Red and Black data processing.

The proposed architecture will be more difficult to analyze than the traditional architecture. However, the analysis task can be reduced to a manageable level by using the Ada language to carefully partitioning the system software so that the Red and Black software routines are logically isolated. In logically isolating these routines we hope to achieve the same type of isolation that is achieved in the traditional architecture, while reducing the parts count and system size by up to a factor of 2.

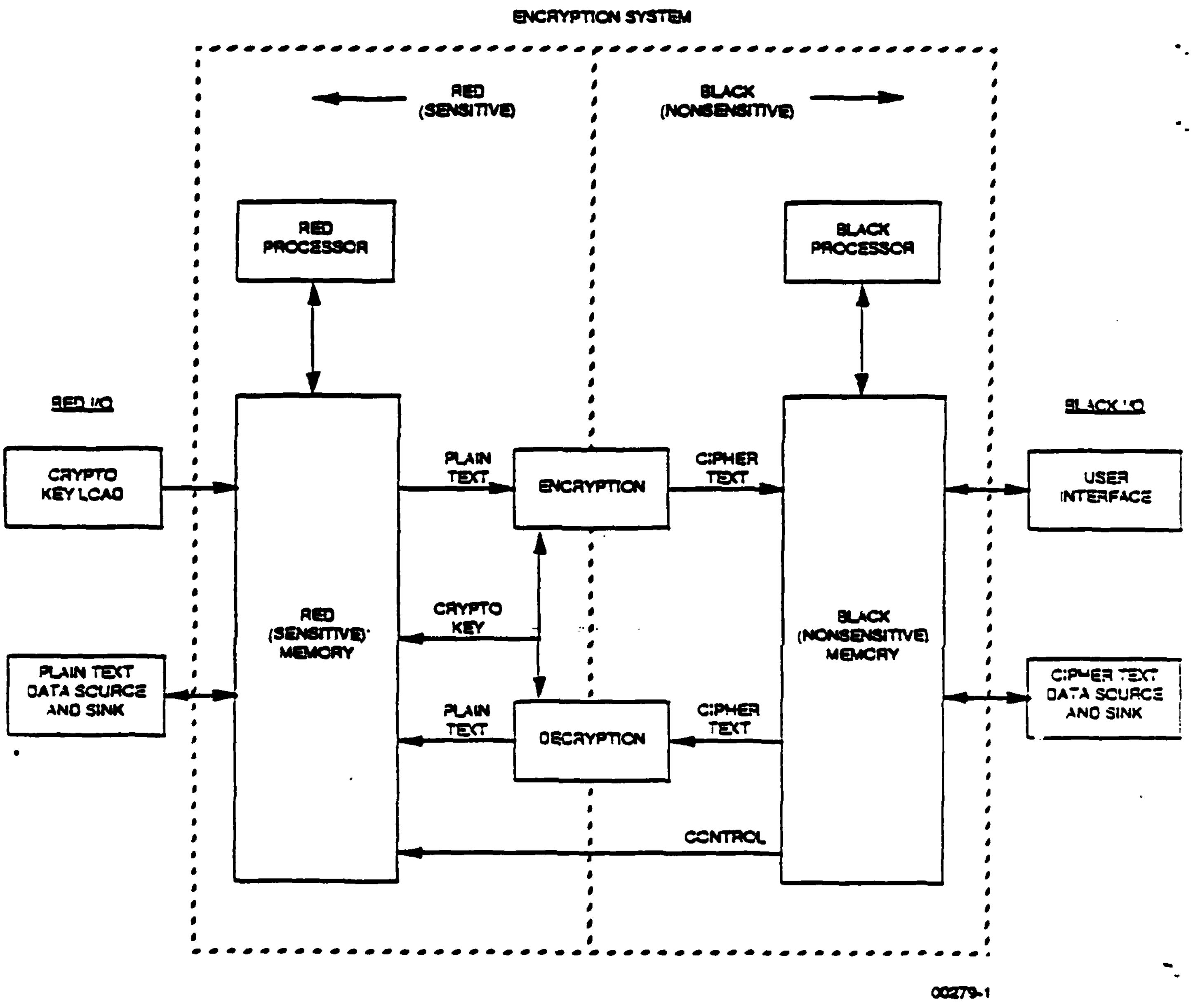


Figure 1 Traditional Encryption System Architecture

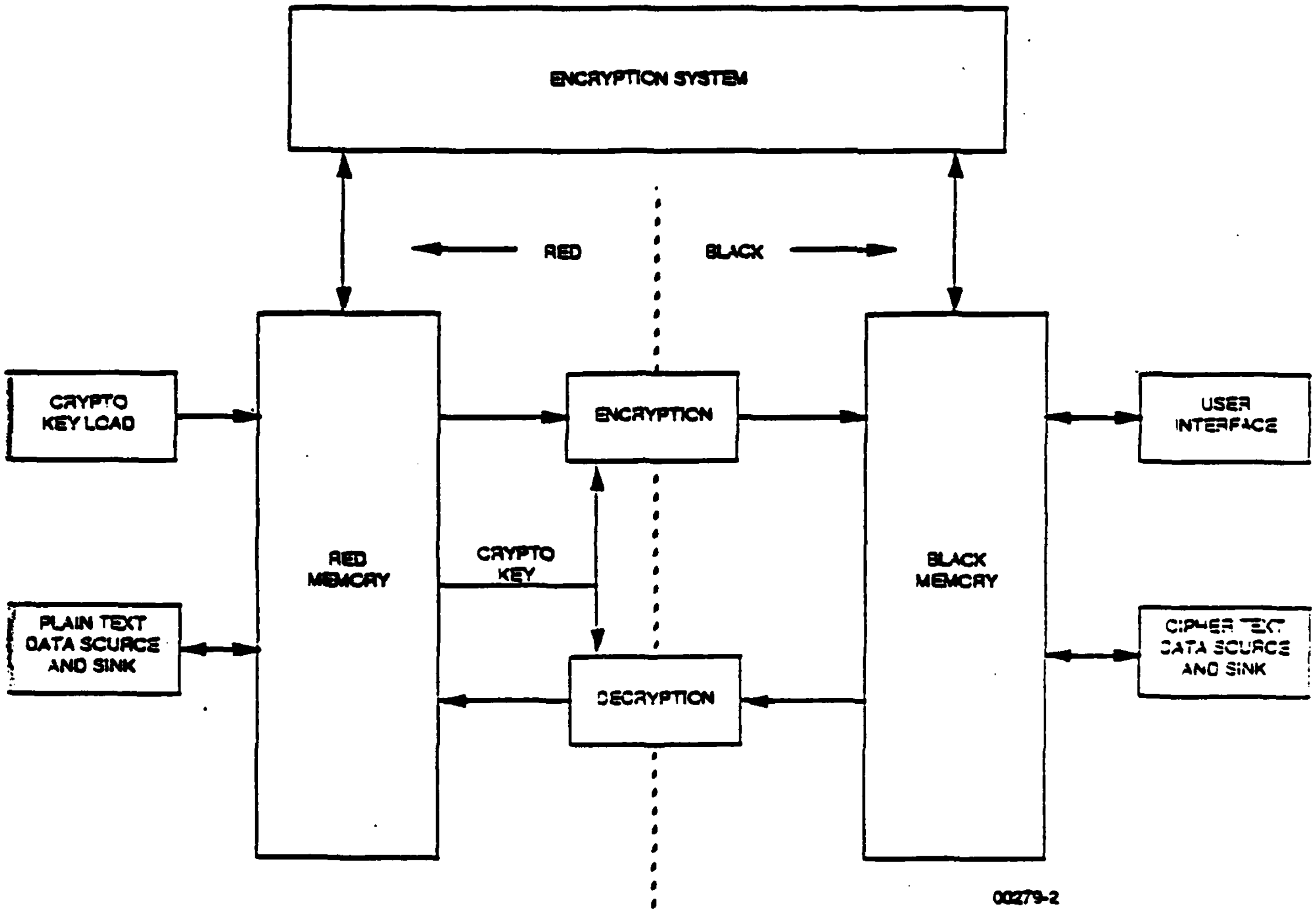


Figure 2 Modified - Encryption System Architecture

Using Ada For Logical Separation

During our work in designing security architectures we have found that we must use a combination of hardware and software to ensure total Red/Black isolation. Ada has helped us achieve this isolation. Some of the software consideration required to achieve isolation include:

- Use of Ada information hiding and package separation to isolate Red and Black routines
- Parameter passing pointers to Red data
- Disabling of interrupts during Red processing routines
- Not using Multi-tasking in secure systems
- Not using boolean type for binary flags
- Using more than one loop counter in software loops
- Including processor self-tests prior to processing Red data

In addition to these software considerations it has been necessary to implement several hardware functions which support the software. These include :

- Use of the User/Supervisor capabilities of processor to prevent black routines from accessing Red data.
- Using physically isolated Red and Black memories
- Using external checking hardware to verify the operation of the processor

In this paper we will concentrate on the software techniques and how Ada will can be used to help ensure logical separation. We will discuss the various hardware techniques only briefly.

Ada's capability to do "information hiding" has proven to be an excellent tool in providing Red/Black separation. Using information hiding, the designer can create a single Ada package which performs all of the Red data handling and processing. This package (we will call it the Red Data Handling Package) must hide all Red data from all other packages in the program. By hiding all the Red data from other packages it prevents these other packages from accidentally mistaking Red data as black data.

The Red Data Handling package should be the only package which has access to the Red RAM. This access control can be enhanced by using the supervisor state of the processor. If we design the hardware so that only the Red data handling package is run in the supervisor mode then we can add hardware which disables the Red RAM except when the processor is in the supervisor state. This will also help to ensure that all packages other than the Red Data Handler Package are physically denied access to Red RAM.

Data can be exchanges between the Red and Black packages only through the encryption function. It is the responsibility of the Red Data Handler to send Red data to and receive Red data from the encryption function.

Isolating all Red data handling operation in the Red Data Handler package also simplifies the analysis function. This is because only routines which have access to Red data need to be analyzed and only the routines in the Red Data Handler package have access to Red data. If the Red Data Handler package is kept small (less than 4000 lines of code) then the analysis task will be simplified.

Although using Ada's information hiding capabilities is a big step towards data separation it is not sufficient to guarantee separation. There are several other accidental ways that Red and Black data can be mixed. One of the major accidental mixings can occur when one Red Data Handler routine is passing red data as a parameter to another Red Data Handler routine. In general, Ada will be using the Black RAM for stack. If the calling routine were to pass the Red data by value then the Ada compiler will place the Red data on the stack (Black RAM) prior to calling the destination routine. In this case, the Ada compiler has inadvertently written the Red data into Black RAM. To prevent this from happening the software engineer must pass all parameters as pointers to Red RAM instead of passing the parameters by value.

Another requirement is to disable all interrupts when in the Red Data Handling package. This requirement is necessary to allow the code to be analyzed. It is difficult enough to try to analyze code when you know the exact sequence of code execution. It is impossible to analyze code if there is a possibility that the sequence of execution will be changed at by an interrupt.

Like interrupts, multi-tasking in a secure environment should also be avoided. If multi-tasking were to be used, a security analysis of the Ada Run-Time System would be required to verify proper operation. The run time system in much too complex to be analyzed. Resources available for evaluating software on a secure encryption system are limited and it is doubtful that more than 4,000 lines of code could be evaluated.

Another concern in a secure system is that the software is operating correctly on the hardware. The software should contain as many checks on the hardware as possible to ensure that the hardware is not malfunctioning. Two simple checks that can be integrated into the Ada code which are the definition of a new boolean type and the use of auxiliary loop counters in all loops.

The Ada boolean type should not be used in secure system since the hamming distance between True and False is only 1. This means that a single bit failure in the boolean flag can cause a True to be interpreted as a false or a vice versa. It is much better to define a new boolean type which assigns 00H to True and FFH to False. This provides a hamming distance of 8 which will prevent single bit errors from causing improper operation. In addition, the Ada range checking option should be enabled to verify that boolean variable can only take on the True and False values and that all other values will cause an exception.

Auxiliary loop counters should also be used to ensure that a single bit failure in a loop variable will not cause improper operation. By using two counters for each loop, the original loop counter and an auxiliary loop counter, and comparing these counters at the end of the loop, single bit error can be detected.

Ada Example

An Ada package using only machine code insertions can be developed to implement the Red Data Handler Package. The use of machine code insertions will guarantee that Red Data is not stored in variables which would require the stack if passed as a parameter. All parameters can be passed using pointers or the processors internal registers.

The use of object oriented design can associate particular functions to areas in the Red RAM used to store different types of data. All of these functions would have the requirements identified for a Red Processing System imposed on them. For example, the Red RAM location identified for holding the common key may have the following functions associated with it in a subpackage:

Receive_From_Red_Input_Port (Red_Input_Type)
Transfer_To_Encryption_Engine (Algorithm_Type)
Transfer_From_Decryption_Engine (Algorithm_Type)
Parity_Check
Clear

The Red RAM location identified for holding Plain_Text may have the following functions associated with it in a subpackage:

Receive_from_Red_Input_Port (Red_Input_Type)	
Encrypt (Algorithm_Type)	— Sends to Plain_Text.Encrypt
Accept (Plain_Text_Type)	— Receive from Cipher_Text.Decrypt
Transmit_To_Red_Output_Port (Red_Output_Type)	
Clear	

Black RAM used by the Red Data Handler does not need Ada machine code insertions. Object oriented design techniques and packaging would still be used in order to encapsulate data and promote information hiding. Functions associated with encrypted cipher text data can be packaged as follows:

Accept (Cipher_Text_Type)	— Transferred from Black system or Plaintext.Encrypt
Transfer_To_Black	— Transfer to Black Processing System
Decrypt (Algorithm_Type)	— Decrypts and transfers to Plain_Text.Accept

CONCLUSION:

A combination of hardware and software techniques are required for a single processor implementation of a Red Processing System. Some of the requirements include:

- Information hiding and Ada Package definitions should be used to logically isolate Red and Black processing routines.
- Physically separate Red and Black memory should be utilized in the system
- The supervisor state of the processor should be used to enable Red RAM operations. Only the Red data handling package should run in the supervisor mode.
- Red data parameters should be passed by address and not by value to prevent Red data form being placed in Black memory
- Interrupts and multi-tasking should not be used in secure systems.
- Care should be taken to use a hamming distance of 2 or more for all loop counters and boolean functions.
- Processor self-test should be performed to verify proper processor operation prior to handling Red data.

For a single processor implementation of a Red Processing System. Ada provides enough of the resources required for a high order language (machine code insertions, information hiding and package constructs) in order to implement a secure single processor design for the Red Processing section of an encryption system.

This paper lists only some of the techniques which can be used when designing secure systems. In general, the security level of the system will dictate which methods are applicable and which methods are not.

POSITION PAPER

PARTIAL VERIFICATION -- A PRACTICAL APPROACH

Trtin Marietta Information
& Communications Systems
P.O. Box 1260
Denver, CO 80201-1260

1. Concept

Although verification technology has existed in the research community for many years, the principal use has been limited to exercises that formally prove software correct. Wider acceptance has been inhibited by the perception that the effort involved in proving software correct can be orders of magnitude greater than the effort to deveation
process to the consideration of selected properties. The existing technology allows one to define properties of a piece of software and formally specify, through assertions within an annotation system, such properties. The effect of such a process is to use verification technology, in the statistical sense, to disprove the hypothesis that the intended use of a piece of software is consistent with a specified set of conditions. As with any statistical approach, one cannot prove that a piece of softwarmentally tests and verifies that
the software is not incorrectly used, one builds more confidence that the software is, indeed, correctly used. In such a context, the use of verification technology is practical and economically feasible.

An annotation system used in this manner is an example of the use of a formal method. The motivating property of a formal method that is implicit in this approach is the production of formal descriptions that can be read and interpreted by software tools. High ordeable specifications. Thus
another benefit of this approach is that it leads to the formation of an executable specification langauge.

2. Application of the Concept

A partial verification approach based upon Ada as the implementation language can be developed from existing tools. Such tools include an Ada compiler, Anna as the annotation language, and the tool set developed for Anna.

The concept can be applied to a library of appropriately annotated Ada code modules, so that, when a particularaappropriate context. The appropriate annotations are assertions. When the Anna tools are employed in the compilation process, these assertions generate code to raise an Anna exception, at run time, if an attempt is made to use the module in an incorrect context.

By concentrating on annotations that are contained within the library modules, the problem that program developers find annotations difficult to develop is alleviated. A consequence of this approach is that the use of annotations does not buonstraints on the Ada model that inhibit or enhance portability.
In this respect, we can investigate how to identify design choices, by Ada compiler implementors, that affect the portability of code.

Similarly, we can annotate design choices by hardware designers that affect the portability of Ada code, and thereby avoid incorrect use of Ada software modules dependent on a missing hardware design feature.

The objective of this use of annotation is to capture information and thus be able to derive confidne architecture.

3. Examples

The use of annotation adds integrity and security to software that contains the following types of modules:

a. An elementary function, such as the Sine function, where a fast version devoid of range reduction could be annotated and used in certain contexts.

b. A subprogram that is applicable only to a numeric subtype that, given the language type model, is not expressible except by annotation. For as byte order. A desired abstraction, most significant byte, can be annotated, while the programming language permits only the abstraction, low-numbered or high-numbered byte. The correspondence between the desired abstraction and the abstraction permitted by the programming language is machine dependent.

d. An operating system dependency, such as a naming convention that impacts the source code.

APPENDIX B

ANALYSIS OF CATALOG OF INTERFACE FEATURES AND OPTIONS (CIFO) (Not Part of the Workshop Outputs)

The analysis summarized in this appendix resulted from an evening meeting that was not part of the workshop. This appendix was made available for informational purposes to the workshop participants. To obtain a copy of Appendix B, contact:

Fred Maymir-Ducharme, Ph.D.
Chair, ARTEWG Security Task Force
IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

Telephone: (301) 731-8894

APPENDIX C
PARTICIPANTS

Dock Allen
Control Data Corporation
HQ6539
P.O. Box 609
Bloomington, MN 55440

James P. Alstad
Hughes Aircraft Company
Support Software Department
P.O. Box 92428
Bldg. R11, MS 10046
Los Angeles, CA 90009

Mary S. Armstrong
IIT Research Institute
4600 Forbes Blvd.
Lanham, Maryland 20706

Edward Beaver
Westinghouse ESG
P.O. Box 746
M.S. 5370
Baltimore, Maryland 21203

George Buchanan
IIT Research Institute
4600 Forbes Blvd.
Lanham, Maryland 20706

Paul M. Cohen
Martin Marietta Information
& Communications Systems
P.O. Box 1260 MS XL1640
Denver, CO 80201-1260

Michael Diaz
Motorola GEG
MS H 1101
8201 East McDowell Road
Scottsdale, Arizona 85252

Douglas Ferguson
Westinghouse ESG
P.O. Box 746
M.S. 5370
Baltimore, Maryland 21203

Clareance "Jay" Ferguson
National Security Agency
9800 Savage Road
Fort George G. Meade, Maryland 20755-6000

Edward Gallagher
US Army CECOM
AMSEL-RD-SE-AST-SS
Fort Monmouth, NJ 07703

Steve Goldstein
IIT Research Institute
4600 Forbes Blvd.
Lanham, Maryland 20706

Jeffrey L. Grover
Manager, LHX-PMO (GTRI)
ERB/Rm 173
Georgia Tech Research Institute
Atlanta, Georgia 30332

Mark Kraieski
MCAir/LHX
5000 E McDowell
Mesa, AZ

Sue LeGrand
Planning Research Corp.
Suite 200
2200 Space Park Drive
Houston, Texas 77058

Nina Lewis
Unisys Corporation
Defense Systems
5151 Camino Ruiz
Camanillo, CA 93010

Ann Marmor-Squires
TRW
Federal Systems Group
2750 Prosperity Avenue
Fairfax, Virginia 22031

Fred Maymir-Ducharme, Ph.D.
IIT Research Institute
4600 Forbes Blvd.
Lanham, Maryland 20706

John McHugh, Ph.D.
Computational Logic, Inc.
3500 Westgate Drive, Suite 204
Durham, North Carolina 27707

John A. Perkins
Dynamics Research Corporation
Systems Division
60 Frontage Road
Andover, Ma 01810

Charles W. McKay, Ph.D.
Software Engineering Research Center
High Technologies Lab
2700 Bay Area Blvd.
Houston, TX 77058-1068

Captain Robert Pierce
AFCSC/SRVC
Bldg. 2012
San Antonio, Texas

Richard Powers
Texas Instruments Defense Systems
and Electronics Group
P.O. Box 869305
M/S 8503
Plano, Texas 75086

Ken Rowe
National Computer Security Center
9800 Savage Road
Ft. George Meade, Maryland 20755-6000

Jonathan C. Shultis, Ph.D.
Incremental Systems Corporation
319 South Craig Street
Pittsburgh, PA 15213

William R. Worger
Motorola, GEG
8201 E. McDowell Road
P.O. Box 1417
Scottsdale, AZ 85252